

# An implementation of ROBDDs for Isabelle/HOL

Julius Michaelis and Maximilian Haslbeck and Peter Lammich and Lars Hupel

October 11, 2017

## Abstract

We present a verified and executable implementation of ROBDDs in Isabelle/HOL. Our implementation relates pointer-based computation in the Heap monad to operations on an abstract definition of boolean functions. Internally, we implemented the if-then-else combinator in a recursive fashion, following the Shannon decomposition of the argument functions. The implementation mixes and adapts known techniques and is built with efficiency in mind.

## Contents

<b>1</b>	<b>Preface</b>	<b>2</b>
<b>2</b>	<b>Boolean functions</b>	<b>2</b>
2.1	Shannon decomposition . . . . .	3
<b>3</b>	<b>Binary Decision Trees</b>	<b>4</b>
<b>4</b>	<b>Option Helpers</b>	<b>21</b>
<b>5</b>	<b>Abstract ITE Implementation</b>	<b>22</b>
<b>6</b>	<b>Pointermap</b>	<b>33</b>
<b>7</b>	<b>Functional interpretation for the abstract implementation</b>	<b>36</b>
<b>8</b>	<b>Array List</b>	<b>44</b>
<b>9</b>	<b>Imparative implementation for Pointermap</b>	<b>45</b>
<b>10</b>	<b>Imparative implementation</b>	<b>47</b>
10.1	A standard library of functions . . . . .	55
10.2	Printing . . . . .	56
<b>11</b>	<b>Collapsing the levels</b>	<b>57</b>

## 1 Preface

This work is not the first to deal with BDDs in Isabelle/HOL. Ortner and Schirmer have formalized BDDs in [4] and proved the correctness of an algorithm that transforms arbitrary BDDs to ROBDDs. However, their specification does not provide efficiently executable algorithms on BDDs. Giorgino and Strecker have presented efficiently executable algorithms for ROBDDs [2] by reducing their arguments to manipulating edges of graphs. However, they have, to the best of our knowledge, not made their theory files available. Thus, no library for efficient computation on (RO)BDDs in Isabelle/HOL existed. Our work is a response to that situation.

The theoretic background of the implementation is mostly based on [1].

## 2 Boolean functions

```
theory Bool-Func
imports Main
begin
```

The end result of our implementation is verified against these functions:

```
type-synonym 'a boolfunc = ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool
```

if-then-else on boolean functions.

```
definition bf-ite i t e  $\equiv$  ( $\lambda$ l. if i l then t l else e l)
```

if-then-else is interesting because we can, together with constant true and false, represent all binary boolean functions using maximally two applications of it.

```
abbreviation bf-True  $\equiv$  ( $\lambda$ l. True)
```

```
abbreviation bf-False  $\equiv$  ( $\lambda$ l. False)
```

A quick demonstration:

```
definition bf-and a b  $\equiv$  bf-ite a b bf-False
```

```
lemma (bf-and a b) as  $\longleftrightarrow$  a as  $\wedge$  b as unfolding bf-and-def bf-ite-def by meson
```

```
definition bf-not b  $\equiv$  bf-ite b bf-False bf-True
```

```
lemma bf-not-alt: bf-not a as  $\longleftrightarrow$   $\neg$ a as unfolding bf-not-def bf-ite-def by meson
```

For convenience, we want a few functions more:

```
definition bf-or a b  $\equiv$  bf-ite a bf-True b
```

```
definition bf-lit v  $\equiv$  ( $\lambda$ l. l v)
```

```
definition bf-if v t e  $\equiv$  bf-ite (bf-lit v) t e
```

**lemma** *bf-if-alt*:  $bf\text{-if } v \ t \ e = (\lambda l. \text{if } l \text{ then } t \ l \ \text{else } e \ l)$  **unfolding** *bf-if-def bf-ite-def bf-lit-def ..*

**definition** *bf-nand*  $a \ b = bf\text{-not } (bf\text{-and } a \ b)$

**definition** *bf-nor*  $a \ b = bf\text{-not } (bf\text{-or } a \ b)$

**definition** *bf-biimp*  $a \ b = (bf\text{-ite } a \ b \ (bf\text{-not } b))$

**lemma** *bf-biimp-alt*:  $bf\text{-biimp } a \ b = (\lambda l. \ a \ l \ \longleftrightarrow \ b \ l)$  **unfolding** *bf-biimp-def bf-not-def bf-ite-def* **by** (*simp add: fun-eq-iff*)

**definition** *bf-xor*  $a \ b = bf\text{-not } (bf\text{-biimp } a \ b)$

**lemma** *bf-xor-alt*:  $bf\text{-xor } a \ b = (bf\text{-ite } a \ (bf\text{-not } b) \ b)$

**unfolding** *bf-xor-def bf-biimp-def bf-not-def*

**unfolding** *bf-ite-def*

**by** *simp*

All of these are implemented and had their implementation verified.

**definition** *bf-imp*  $a \ b = bf\text{-ite } a \ b \ bf\text{-True}$

**lemma** *bf-imp-alt*:  $bf\text{-imp } a \ b = bf\text{-or } (bf\text{-not } a) \ b$  **unfolding** *bf-or-def bf-not-def bf-imp-def* **unfolding** *bf-ite-def* **unfolding** *fun-eq-iff* **by** *simp*

**lemma** [*dest!, elim!*]:  $bf\text{-False} = bf\text{-True} \implies \text{False}$   $bf\text{-True} = bf\text{-False} \implies \text{False}$  **unfolding** *fun-eq-iff* **by** *simp-all*

**lemmas** [*simp*] = *bf-and-def bf-or-def bf-nand-def bf-biimp-def bf-xor-alt bf-nor-def bf-not-def*

## 2.1 Shannon decomposition

A restriction of a boolean function on a variable is creating the boolean function that evaluates as if that variable was set to a fixed value:

**definition** *bf-restrict*  $(i::'a) \ (val::bool) \ (f::'a \ \text{bool} \ \text{func}) \equiv (\lambda v. \ f \ (v(i:=val)))$

Restrictions are useful, because they remove variables from the set of significant variables:

**definition** *bf-vars*  $bf = \{v. \ \exists as. \ bf\text{-restrict } v \ \text{True } bf \ as \neq \ bf\text{-restrict } v \ \text{False } bf \ as\}$

**lemma**  $var \notin bf\text{-vars} \ (bf\text{-restrict } var \ val \ ex)$

**unfolding** *bf-vars-def bf-restrict-def* **by** (*simp*)

We can decompose calculating if-then-else into computing if-then-else of two triples of functions with one variable restricted to true / false. Given that the functions have finite arity, we can use this to construct a recursive definition.

**lemma** *brace90shannon*:  $bf\text{-ite } F \ G \ H \ ass =$

$bf\text{-ite } (\lambda l. \ l \ i)$

$(bf\text{-ite } (bf\text{-restrict } i \ \text{True } F) \ (bf\text{-restrict } i \ \text{True } G) \ (bf\text{-restrict } i \ \text{True } H))$

$(bf\text{-ite } (bf\text{-restrict } i \ \text{False } F) \ (bf\text{-restrict } i \ \text{False } G) \ (bf\text{-restrict } i \ \text{False } H))$

*ass*

**unfolding** *bf-ite-def bf-restrict-def* **by** (*auto simp add: fun-upd-idem*)

end

### 3 Binary Decision Trees

```
theory BDT
imports Bool-Func
begin
```

We first define all operations and properties on binary decision trees. This has the advantage that we can use a simple, structurally defined type and the disadvantage that we cannot represent sharing.

```
datatype 'a ifex = Trueif | Falseif | IF 'a 'a ifex 'a ifex
```

The type is the same as in Boolean Expression Checkers by Nipkow [3]. Internally, Boolean Expression Checkers transforms the boolean expressions to reduced BDTs of this type. Tests like being tautology testing are then trivial.

```
fun val-ifex :: 'a ifex  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool where
  val-ifex Trueif s = True |
  val-ifex Falseif s = False |
  val-ifex (IF n t1 t2) s = (if s n then val-ifex t1 s else val-ifex t2 s)
```

```
fun ifex-vars :: ('a :: linorder) ifex  $\Rightarrow$  'a list where
  ifex-vars (IF v t e) = v # ifex-vars t @ ifex-vars e |
  ifex-vars Trueif = [] |
  ifex-vars Falseif = []
```

```
abbreviation ifex-var-set a  $\equiv$  set (ifex-vars a)
```

```
fun ifex-ordered :: ('a :: linorder) ifex  $\Rightarrow$  bool where
  ifex-ordered (IF v t e) = (( $\forall$  tv  $\in$  (ifex-var-set t  $\cup$  ifex-var-set e). v < tv)
     $\wedge$  ifex-ordered t  $\wedge$  ifex-ordered e) |
  ifex-ordered Trueif = True |
  ifex-ordered Falseif = True
```

```
fun ifex-minimal :: ('a :: linorder) ifex  $\Rightarrow$  bool where
  ifex-minimal (IF v t e)  $\longleftrightarrow$  t  $\neq$  e  $\wedge$  ifex-minimal t  $\wedge$  ifex-minimal e |
  ifex-minimal Trueif = True |
  ifex-minimal Falseif = True
```

```
abbreviation ro-ifex where ro-ifex t  $\equiv$  ifex-ordered t  $\wedge$  ifex-minimal t
```

```
definition bf-ifex-rel where
  bf-ifex-rel = {(a,b). ( $\forall$  ass. a ass  $\longleftrightarrow$  val-ifex b ass)  $\wedge$  ro-ifex b}
```

```
lemma ifex-var-noinfluence: x  $\notin$  ifex-var-set b  $\implies$  val-ifex b (ass(x:=val)) =
  val-ifex b ass
```

**by** (*induction b, auto*)

**lemma** *roifex-var-not-in-subtree*:

**assumes** *ro-ifex b* **and**  $b = IF\ v\ t\ e$

**shows**  $v \notin ifex\text{-var}\text{-set}\ t$  **and**  $v \notin ifex\text{-var}\text{-set}\ e$

**using** *assms* **by** (*induction, auto*)

**lemma** *roifex-set-var-subtree*:

**assumes** *ro-ifex b* **and**  $b = IF\ v\ t\ e$

**shows**  $val\text{-ifex}\ b\ (ass(v:=True)) = val\text{-ifex}\ t\ ass$

$val\text{-ifex}\ b\ (ass(v:=False)) = val\text{-ifex}\ e\ ass$

**using** *assms* **by** (*auto intro!: ifex-var-noinfluence dest: roifex-var-not-in-subtree*)

**lemma** *roifex-Trueif-unique*:  $ro\text{-ifex}\ b \implies \forall\ ass.\ val\text{-ifex}\ b\ ass \implies b = Trueif$

**proof**(*induction b*)

**case** ( $IF\ v\ b1\ b2$ ) **with** *roifex-set-var-subtree*[*OF*  $\langle ro\text{-ifex}\ (IF\ v\ b1\ b2) \rangle$ ] **show**  
*?case* **by** *force*

**qed**(*auto*)

**lemma** *roifex-Falseif-unique*:  $ro\text{-ifex}\ b \implies \forall\ ass.\ \neg\ val\text{-ifex}\ b\ ass \implies b = Falseif$

**proof**(*induction b*)

**case** ( $IF\ v\ b1\ b2$ ) **with** *roifex-set-var-subtree*[*OF*  $\langle ro\text{-ifex}\ (IF\ v\ b1\ b2) \rangle$ , *of v b1 b2*]  
**show** *?case*

**by** *fastforce*

**qed**(*auto*)

**lemma**  $(f, b) \in bf\text{-ifex}\text{-rel} \implies b = Trueif \longleftrightarrow f = (\lambda\ \cdot.\ True)$

**unfolding** *bf-ifex-rel-def* **using** *roifex-Trueif-unique* **by** *auto*

**lemma**  $(f, b) \in bf\text{-ifex}\text{-rel} \implies b = Falseif \longleftrightarrow f = (\lambda\ \cdot.\ False)$

**unfolding** *bf-ifex-rel-def* **using** *roifex-Falseif-unique* **by** *auto*

**lemma** *ifex-ordered-not-part*:  $ifex\text{-ordered}\ b \implies b = IF\ v\ b1\ b2 \implies w < v \implies w \notin ifex\text{-var}\text{-set}\ b$

**using** *less-asym* **by** *fastforce*

**lemma** *ro-ifex-unique*:  $ro\text{-ifex}\ x \implies ro\text{-ifex}\ y \implies (\bigwedge\ ass.\ val\text{-ifex}\ x\ ass = val\text{-ifex}\ y\ ass) \implies x = y$

**proof**(*induction x arbitrary: y*)

**case** ( $IF\ xv\ xb1\ xb2$ ) **note**  $IFind = IF$

**from**  $\langle ro\text{-ifex}\ (IF\ xv\ xb1\ xb2) \rangle$   $\langle ro\text{-ifex}\ y \rangle$   $\langle \bigwedge\ ass.\ val\text{-ifex}\ (IF\ xv\ xb1\ xb2)\ ass = val\text{-ifex}\ y\ ass \rangle$

**show** *?case*

**proof**(*induction y*)

**case** ( $IF\ yv\ yb1\ yb2$ )

**obtain**  $x$  **where**  $x\text{-def}: x = IF\ xv\ xb1\ xb2$  **by** *simp*

**obtain**  $y'$  **where**  $y'\text{-def}: y' = IF\ yv\ yb1\ yb2$  **by** *simp*

**from**  $y'\text{-def}\ x\text{-def}\ IFind\ IF$  **have**  $0: ro\text{-ifex}\ xb1\ ro\text{-ifex}\ xb2\ ro\text{-ifex}\ yb1\ ro\text{-ifex}\ yb2\ ro\text{-ifex}\ x\ ro\text{-ifex}\ y'$  **by** *auto*

```

    from IF IFind x-def y'-def have 1:  $\bigwedge$  ass. val-ifex x ass = val-ifex y' ass
  by simp
    show ?case
      proof(cases xv = yv)
        case True
      have xb1 = yb1
      by (auto intro: IFind simp add: 0 1 True roifex-set-var-subtree[OF - y'-def]
        roifex-set-var-subtree[OF - x-def, symmetric])
      moreover have xb2 = yb2
      by (auto intro: IFind simp add: 0 1 True roifex-set-var-subtree[OF - y'-def]
        roifex-set-var-subtree[OF - x-def, symmetric])
      ultimately show ?thesis using True by simp
  next
  case False note uneq = False show ?thesis
  proof(cases xv < yv)
    case True
      from ifex-ordered-not-part[OF - y'-def True] ifex-var-noinfluence[of xv y'
- True]
        0(6) roifex-set-var-subtree(1)[OF 0(5) x-def] 1
      have 5:  $\bigwedge$  ass. val-ifex xb1 ass = val-ifex x ass by blast
      from 0(5) ifex-var-noinfluence[of xv xb1 - False]
        ifex-var-noinfluence[of xv xb2 - False]
        x-def
      have  $\bigwedge$  ass. val-ifex xb1 (ass(xv := False)) = val-ifex xb1 ass
         $\bigwedge$  ass. val-ifex xb2 (ass(xv := False)) = val-ifex xb2 ass by auto
      from 5 this roifex-set-var-subtree(2)[OF 0(5) x-def]
      have  $\bigwedge$  ass. val-ifex xb1 ass = val-ifex xb2 ass by presburger
      from IFind(1)[OF 0(1) 0(2)] this IFind(3) have False by auto
      from this show ?thesis ..
    next
    case False
      from this uneq have 6: yv < xv by auto
      from ifex-ordered-not-part[OF - x-def this]
        ifex-var-noinfluence[of yv x] 0(5)
      have  $\bigwedge$  ass val. val-ifex x (ass(yv := val)) = val-ifex x ass
         $\bigwedge$  ass val. val-ifex x (ass(yv := val)) = val-ifex x ass by auto
      from this roifex-set-var-subtree[OF 0(5) x-def]
      have  $\bigwedge$  ass val. val-ifex x (ass(xv := True, yv := val)) = val-ifex xb1 ass
         $\bigwedge$  ass val. val-ifex x (ass(xv := False, yv := val)) = val-ifex xb2 ass
  by blast+
  from ifex-ordered-not-part[OF - x-def 6] 0(5) ifex-var-noinfluence[of yv x]
  1
    roifex-set-var-subtree[OF 0(6) y'-def]
  have  $\bigwedge$  ass val. val-ifex x ass = val-ifex yb1 ass
     $\bigwedge$  ass val. val-ifex x ass = val-ifex yb2 ass by blast+
  from this IF(1,2) x-def 0(5) y'-def 0(6) have x = yb1 x = yb2 by
fastforce+
  from this have yb1 = yb2 by auto
  from 0(6) y'-def this have False by auto

```

```

      thus ?thesis ..
    qed
  qed
qed (fastforce intro: roifex-Falseif-unique roifex-Trueif-unique)+
qed (fastforce intro: roifex-Falseif-unique[symmetric] roifex-Trueif-unique[symmetric])+

theorem bf-ifex-rel-single: single-valued bf-ifex-rel single-valued (bf-ifex-rel-1)
  unfolding single-valued-def bf-ifex-rel-def using ro-ifex-unique by auto

lemma bf-ifex-eq: (af, at) ∈ bf-ifex-rel ⇒ (bf, bt) ∈ bf-ifex-rel ⇒ (af = bf)
  ⇔ (at = bt)
  unfolding bf-ifex-rel-def using ro-ifex-unique by auto

lemma nonempty-if-var-set: ifex-vars (IF v t e) ≠ [] by auto

fun restrict where
  restrict (IF v t e) var val = (let rt = restrict t var val; re = restrict e var val in
    (if v = var then (if val then rt else re) else (IF v rt re))) |
  restrict i - - = i

declare Let-def[simp]

lemma not-element-restrict: var ∉ ifex-var-set (restrict b var val)
  by (induction b) auto

lemma restrict-assignment: val-ifex b (ass(var := val)) ⇔ val-ifex (restrict b var val) ass
  by (induction b) auto

lemma restrict-variables-subset: ifex-var-set (restrict b var val) ⊆ ifex-var-set b
  by (induction b) auto

lemma restrict-ifex-ordered-invar: ifex-ordered b ⇒ ifex-ordered (restrict b var val)
  using restrict-variables-subset by (induction b) (fastforce)+

lemma restrict-val-invar: ∀ ass. a ass = val-ifex b ass ⇒
  (bf-restrict var val a) ass = val-ifex (restrict b var val) ass
  unfolding bf-restrict-def using restrict-assignment by simp

lemma restrict-untouched-id: x ∉ ifex-var-set t ⇒ restrict t x val = t
proof(induction t)
  case (IF v t e)
  from IF.premis have x ∉ ifex-var-set t x ∉ ifex-var-set e by simp-all
  note mIH = IF.IH(1)[OF this(1)] IF.IH(2)[OF this(2)]
  from IF.premis have x ≠ v by simp
  thus ?case unfolding restrict.simps Let-def mIH by simp
qed simp-all

```

**fun** *ifex-top-var* :: 'a ifex  $\Rightarrow$  'a option **where**  
*ifex-top-var* (IF v t e) = Some v |  
*ifex-top-var* - = None

**fun** *restrict-top* :: ('a :: linorder) ifex  $\Rightarrow$  'a  $\Rightarrow$  bool  $\Rightarrow$  'a ifex **where**  
*restrict-top* (IF v t e) var val = (if v = var then (if val then t else e) else (IF v t e)) |  
*restrict-top* i - - = i

**lemma** *restrict-top-id*: ifex-ordered e  $\Longrightarrow$  ifex-top-var e = Some v  $\Longrightarrow$  v' < v  $\Longrightarrow$   
*restrict-top* e v' val = e  
**by**(*induction* e) *auto*

**lemma** *restrict-id*: ifex-ordered e  $\Longrightarrow$  ifex-top-var e = Some v  $\Longrightarrow$  v' < v  $\Longrightarrow$   
*restrict* e v' val = e  
**proof**(*induction* e *arbitrary*: v)  
**case** (IF w e1 e2) **thus** ?case **by** (cases e1; cases e2; force)  
**qed**(*auto*)

**lemma** *restrict-top-IF-id*: ifex-ordered (IF v t e)  $\Longrightarrow$  v' < v  $\Longrightarrow$  *restrict-top* (IF v t e) v' val = (IF v t e)  
**using** *restrict-top-id* **by** *auto*

**lemma** *restrict-IF-id*: **assumes** o: ifex-ordered (IF v t e) **assumes** le: v' < v  
**shows** *restrict* (IF v t e) v' val = (IF v t e)  
**using** *restrict-id*[OF o, *unfolded ifex-top-var.simps*, OF *refl le*, of val] .

**lemma** *restrict-top-eq*: ifex-ordered (IF v t e)  $\Longrightarrow$  *restrict* (IF v t e) v val =  
*restrict-top* (IF v t e) v val  
**using** *restrict-untouched-id* **by** *auto*

**lemma** *restrict-top-ifex-ordered-invar*: ifex-ordered b  $\Longrightarrow$  ifex-ordered (*restrict-top* b var val)  
**by** (*induction* b) *simp-all*

**fun** *lowest-tops* :: ('a :: linorder) ifex list  $\Rightarrow$  'a option **where**  
*lowest-tops* [] = None |  
*lowest-tops* ((IF v - -)#r) = Some (case *lowest-tops* r of Some u  $\Rightarrow$  (min u v) | None  $\Rightarrow$  v) |  
*lowest-tops* (-#r) = *lowest-tops* r

**lemma** *lowest-tops-NoneD*: *lowest-tops* k = None  $\Longrightarrow$  ( $\neg(\exists v t e. ((IF v t e) \in \text{set } k))$ )  
**by** (*induction* k *rule*: *lowest-tops.induct*) *simp-all*

**lemma** *lowest-tops-in*: *lowest-tops* k = Some l  $\Longrightarrow$  l  $\in$  set (concat (map *ifex-vars* k))  
**by**(*induction* k *rule*: *lowest-tops.induct*) (*simp-all split*: *option.splits if-splits add*:



*min-def*)

**definition**  $IFC\ v\ t\ e \equiv (if\ t = e\ then\ t\ else\ IF\ v\ t\ e)$

**function**  $ifex\ ite :: 'a\ ifex \Rightarrow 'a\ ifex \Rightarrow 'a\ ifex \Rightarrow ('a :: linorder)\ ifex$  **where**  
 $ifex\ ite\ i\ t\ e = (case\ lowest\ tops\ [i,\ t,\ e]\ of\ Some\ x \Rightarrow$   
 $(IFC\ x\ (ifex\ ite\ (restrict\ top\ i\ x\ True)\ (restrict\ top\ t\ x\ True)$   
 $(restrict\ top\ e\ x\ True)))$   
 $(ifex\ ite\ (restrict\ top\ i\ x\ False)\ (restrict\ top\ t\ x\ False)$   
 $(restrict\ top\ e\ x\ False)))$   
 $| None \Rightarrow (case\ i\ of\ True\ if \Rightarrow t\ | False\ if \Rightarrow e))$   
**by** *pat-completeness auto*

**lemma** *restrict-size-le*:  $size\ (restrict\ top\ k\ var\ val) \leq size\ k$   
**by** (*induction k, auto*)

**lemma** *restrict-size-less*:  $ifex\ top\ var\ k = Some\ var \implies size\ (restrict\ top\ k\ var\ val) < size\ k$   
**by** (*induction k, auto*)

**lemma** *lowest-tops-cases*:  
 $lowest\ tops\ [i,\ t,\ e] = Some\ var \implies ifex\ top\ var\ i = Some\ var \vee ifex\ top\ var\ t$   
 $= Some\ var \vee ifex\ top\ var\ e = Some\ var$   
**by** (*(cases i; cases t; cases e), auto simp add: min-def*)

**lemma** *lowest-tops-lowest*:  $lowest\ tops\ es = Some\ a \implies e \in set\ es \implies ifex\ ordered\ e \implies v \in ifex\ var\ set\ e \implies a \leq v$

**proof** (*induction arbitrary: a rule: lowest-tops.induct*)  
**case** 2 **thus** ?*case*  
**proof** (*cases e*)  
**case** *IF* **with** 2 **show** ?*thesis*  
**apply** (*simp add: min-def Ball-def less-imp-le split: if-splits option.splits*)  
**apply** (*meson less-imp-le lowest-tops-NoneD order-refl*)  
**by** *fastforce+*  
**qed** *simp+*  
**qed** *fastforce+*

**lemma** *termlemma2*:  $lowest\ tops\ [i,\ t,\ e] = Some\ xa \implies$   
 $(size\ (restrict\ top\ i\ xa\ val) + size\ (restrict\ top\ t\ xa\ val) + size\ (restrict\ top\ e\ xa\ val)) <$   
 $(size\ i + size\ t + size\ e)$   
**using** *restrict-size-le[of i xa val] restrict-size-le[of t xa val] restrict-size-le[of e xa val]*  
**by** (*auto dest!: lowest-tops-cases restrict-size-less[of - - val]*)

**lemma** *termlemma*:  $lowest\ tops\ [i,\ t,\ e] = Some\ xa \implies$   
 $(case\ (restrict\ top\ i\ xa\ val,\ restrict\ top\ t\ xa\ val,\ restrict\ top\ e\ xa\ val)\ of$   
 $(i,\ t,\ e) \Rightarrow size\ i + size\ t + size\ e) <$   
 $(case\ (i,\ t,\ e)\ of\ (i,\ t,\ e) \Rightarrow size\ i + size\ t + size\ e)$

**using** *termlemma2* **by** *fast*

**termination** *ifex-ite*

**by** (*relation measure*  $(\lambda(i,t,e). \text{size } i + \text{size } t + \text{size } e)$ , *rule* *wf-measure*, *unfold in-measure*)

(*simp-all only: termlemma*)

**definition** *const x - = x*

**declare** *const-def*[*simp*]

**lemma** *rel-true-false*:  $(a, \text{Trueif}) \in \text{bf-ifex-rel} \implies a = \text{const True}$   $(a, \text{Falseif}) \in \text{bf-ifex-rel} \implies a = \text{const False}$

**unfolding** *fun-eq-iff const-def*

**unfolding** *bf-ifex-rel-def*

**by** *simp-all*

**lemma** *rel-if*:  $(a, \text{IF } v \text{ } t \text{ } e) \in \text{bf-ifex-rel} \implies (ta, t) \in \text{bf-ifex-rel} \implies (ea, e) \in \text{bf-ifex-rel} \implies a = (\lambda as. \text{if } as \text{ } v \text{ then } ta \text{ } as \text{ else } ea \text{ } as)$

**unfolding** *fun-eq-iff const-def*

**unfolding** *bf-ifex-rel-def*

**by** *simp-all*

**lemma** *ifex-ordered-implied*:  $(a, b) \in \text{bf-ifex-rel} \implies \text{ifex-ordered } b$  **unfolding** *bf-ifex-rel-def* **by** *simp*

**lemma** *ifex-minimal-implied*:  $(a, b) \in \text{bf-ifex-rel} \implies \text{ifex-minimal } b$  **unfolding** *bf-ifex-rel-def* **by** *simp*

**lemma** *ifex-ite-induct2*[*case-names Trueif Falseif IF*]:

$(\bigwedge i \text{ } t \text{ } e. \text{lowest-tops } [i, t, e] = \text{None} \implies i = \text{Trueif} \implies \text{sentence } i \text{ } t \text{ } e) \implies$

$(\bigwedge i \text{ } t \text{ } e. \text{lowest-tops } [i, t, e] = \text{None} \implies i = \text{Falseif} \implies \text{sentence } i \text{ } t \text{ } e) \implies$

$(\bigwedge i \text{ } t \text{ } e. \text{sentence } (\text{restrict-top } i \text{ } a \text{ } \text{True}) (\text{restrict-top } t \text{ } a \text{ } \text{True}) (\text{restrict-top } e \text{ } a \text{ } \text{True})) \implies$

$\text{sentence } (\text{restrict-top } i \text{ } a \text{ } \text{False}) (\text{restrict-top } t \text{ } a \text{ } \text{False}) (\text{restrict-top } e \text{ } a \text{ } \text{False}) \implies$

$\text{lowest-tops } [i, t, e] = \text{Some } a \implies \text{sentence } i \text{ } t \text{ } e) \implies \text{sentence } i \text{ } t \text{ } e$

**proof**(*induction* *i t e rule: ifex-ite.induct, goal-cases*)

**case**  $(1 \text{ } i \text{ } t \text{ } e)$  **show** *?case*

**proof**(*cases lowest-tops*  $[i, t, e]$ )

**case** *None* **thus** *?thesis* **by** (*cases* *i*) (*auto intro: 1*)

**next**

**case**  $(\text{Some } a)$  **thus** *?thesis* **by**(*auto intro: 1*)

**qed**

**qed**

**lemma** *ifex-ite-induct*[*case-names Trueif Falseif IF*]:

$(\bigwedge i \text{ } t \text{ } e. \text{lowest-tops } [i, t, e] = \text{None} \implies i = \text{Trueif} \implies P \text{ } i \text{ } t \text{ } e) \implies$

$(\bigwedge i \text{ } t \text{ } e. \text{lowest-tops } [i, t, e] = \text{None} \implies i = \text{Falseif} \implies P \text{ } i \text{ } t \text{ } e) \implies$

```

( $\wedge i t e a. (\wedge val. P (restrict-top i a val) (restrict-top t a val) (restrict-top e a val)) \implies$ 
  lowest-tops  $[i, t, e] = Some a \implies P i t e \implies P i t e$ 
proof(induction  $i t e$  rule: ifex-ite-induct2)
  case (IF  $i t e a$ )
  have  $\wedge val. (P (restrict-top i a val) (restrict-top t a val) (restrict-top e a val))$ 
    by (case-tac val) (clarsimp, blast intro: IF)+
  with IF show ?case by blast
qed blast+

```

```

lemma restrict-top-subset:  $x \in ifex-var-set (restrict-top i vr vl) \implies x \in ifex-var-set i$ 
by(induction  $i$ ) (simp-all split: if-splits)

```

```

lemma ifex-vars-subset:  $x \in ifex-var-set (ifex-ite i t e) \implies (x \in ifex-var-set i) \vee$ 
 $(x \in ifex-var-set t) \vee (x \in ifex-var-set e)$ 
proof(induction rule: ifex-ite-induct2)
  case (IF  $i t e a$ )
  have  $x \in \{x. x = a\} \vee x \in (ifex-var-set (ifex-ite (restrict-top i a True) (restrict-top t a True) (restrict-top e a True))) \vee x \in (ifex-var-set (ifex-ite (restrict-top i a False) (restrict-top t a False) (restrict-top e a False)))$ 
    using IF by(simp add: IFC-def split: if-splits)
  hence  $x = a \vee$ 
 $x \in (ifex-var-set (restrict-top i a True)) \vee x \in (ifex-var-set (restrict-top t a True)) \vee x \in (ifex-var-set (restrict-top e a True)) \vee$ 
 $x \in (ifex-var-set (restrict-top i a False)) \vee x \in (ifex-var-set (restrict-top t a False)) \vee x \in (ifex-var-set (restrict-top e a False))$ 
    using IF by blast
  thus ?case
    using restrict-top-subset apply -
    apply(erule disjE)
    subgoal using lowest-tops-in[OF IF(3)] apply(simp only: set-concat set-map set-simps) by blast
    by blast
qed simp-all

```

```

lemma three-ins:  $i \in set [i, t, e] t \in set [i, t, e] e \in set [i, t, e]$  by simp-all

```

```

lemma hlp3: lowest-tops (IF  $v uu uv \# r$ )  $\neq$  lowest-tops  $r \implies$  lowest-tops (IF  $v uu uv \# r$ ) = Some  $v$ 
by(simp add: min-def split: option.splits if-splits)

```

```

lemma hlp2: IF  $vi vt ve \in set is \implies$  lowest-tops  $is = Some a \implies a \leq vi$ 
apply(induction  $is$  arbitrary:  $vt ve a$  rule: lowest-tops.induct)
  prefer 2
  subgoal
    apply(auto simp add: min-def split: if-splits option.splits dest: lowest-tops-NoneD)
    by (meson le-cases order-trans)
  by (auto)

```

**lemma** *hlp1*:  $i \in \text{set } is \implies \text{lowest-tops } is = \text{Some } a \implies \text{ifex-ordered } i \implies a \notin (\text{ifex-var-set } (\text{restrict-top } i \ a \ \text{val}))$   
**proof**(*rule ccontr, unfold not-not, goal-cases*)  
  **case** 1  
  **from** 1(4) **obtain** *vi vt ve* **where** *vi*:  $i = IF \ vi \ vt \ ve$  **by**(*cases i*) *simp-all*  
  **with** 1 **have** *ne*:  $vi \neq a$  **by**(*simp split: if-splits*) *blast+*  
  **moreover** **have**  $vi \leq a$  **using** 1(3,4) **proof**( $-, \text{goal-cases}$ )  
  **case** 1  
  **hence**  $a \in (\text{ifex-var-set } vt) \vee a \in (\text{ifex-var-set } ve)$  **using** *ne* **by**(*simp add: vi*)  
  **thus** ?*case* **using** (*ifex-ordered i*) *vi* **using** *less-imp-le* **by** *auto*  
  **qed**  
  **moreover** **have**  $a \leq vi$  **using** 1(1) **unfolding** *vi* **using** 1(2) *hlp2* **by** *metis*  
  **ultimately** **show** *False* **by** *simp*  
**qed**

**lemma** *order-ifex-ite-invar*:  $\text{ifex-ordered } i \implies \text{ifex-ordered } t \implies \text{ifex-ordered } e \implies \text{ifex-ordered } (\text{ifex-ite } i \ t \ e)$   
**proof**(*induction i t e rule: ifex-ite-induct*)  
  **case** (*IF i t e*) **note** *goal1 = IF*  
  **note** *l = restrict-top-ifex-ordered-invar*  
  **note**  $l[OF \ \text{goal1}(3)] \ l[OF \ \text{goal1}(4)] \ l[OF \ \text{goal1}(5)]$   
  **note** *mIH = goal1(1)[OF this]*  
  **note** *blubb = lowest-tops-lowest[OF goal1(2) - - restrict-top-subset]*  
  **show** ?*case* **using** *mIH*  
  **by** (*subst ifex-ite.simps,*  
  *auto simp del: ifex-ite.simps*  
  *simp add: IFC-def goal1(2) hlp1[OF three-ins(1) goal1(2) goal1(3)] hlp1[OF*  
  *three-ins(2) goal1(2) goal1(4)] hlp1[OF three-ins(3) goal1(2) goal1(5)]*  
  *dest: ifex-vars-subset blubb[OF three-ins(1) goal1(3)] blubb[OF three-ins(2)*  
  *goal1(4)] blubb[OF three-ins(3) goal1(5)]*  
  *intro!: le-neq-trans*)  
**qed** *simp-all*

**lemma** *ifc-split*:  $P \ (IFC \ v \ t \ e) \iff ((t = e) \longrightarrow P \ t) \wedge (t \neq e \longrightarrow P \ (IF \ v \ t \ e))$   
  **unfolding** *IFC-def* **by** *simp*

**lemma** *restrict-top-ifex-minimal-invar*:  $\text{ifex-minimal } i \implies \text{ifex-minimal } (\text{restrict-top } i \ a \ \text{val})$   
  **by**(*induction i*) *simp-all*

**lemma** *minimal-ifex-ite-invar*:  $\text{ifex-minimal } i \implies \text{ifex-minimal } t \implies \text{ifex-minimal } e \implies \text{ifex-minimal } (\text{ifex-ite } i \ t \ e)$   
  **by**(*induction i t e rule: ifex-ite-induct*) (*simp-all split: ifc-split option.split add: restrict-top-ifex-minimal-invar*)

**lemma** *restrict-top-bf*:  $i \in \text{set } is \implies \text{lowest-tops } is = \text{Some } vr \implies \text{ifex-ordered } i \implies (\bigwedge \text{ass. } fi \ \text{ass} = \text{val-ifex } i \ \text{ass}) \implies \text{val-ifex } (\text{restrict-top } i \ vr \ vl) \ \text{ass} = \text{bf-restrict } vr \ vl \ fi \ \text{ass}$

```

proof(cases i, goal-cases)
  case (3 x31 x32 x33) note goal3 = 3
  have rr: restrict-top i vr vl = restrict i vr vl
  proof(cases x31 = vr)
    case True
    note uf = restrict-top-eq[OF goal3(3)[unfolded goal3(5)], symmetric, unfolded
goal3(5)[symmetric], unfolded True]
    thus ?thesis .
  next
  case False
  have 1: restrict-top i vr vl = i by (simp add: False goal3(5))
  have vr < x31 using le-neg-trans[OF hlp2[OF goal3(1)[unfolded goal3(5)]
goal3(2)] False[symmetric]] by blast
  with goal3(3,5) have 2: restrict i vr vl = i using restrict-IF-id by blast
  show ?thesis unfolding 1 2 ..
  qed
  show ?case unfolding rr by(simp add: goal3(4) restrict-val-invar[symmetric])
qed (simp-all add: bf-restrict-def)

```

**lemma** val-ifex-ite:

```

(∧ ass. fi ass = val-ifex i ass) ⇒
(∧ ass. ft ass = val-ifex t ass) ⇒
(∧ ass. fe ass = val-ifex e ass) ⇒
ifex-ordered i ⇒ ifex-ordered t ⇒ ifex-ordered e ⇒
(bf-ite fi ft fe) ass = val-ifex (ifex-ite i t e) ass

```

**proof**(induction i t e arbitrary: fi ft fe rule: ifex-ite-induct)

**case** (IF i t e a)

**note** mIH = IF(1)[OF refl refl refl

restrict-top-ifex-ordered-invar[OF IF(6)]

restrict-top-ifex-ordered-invar[OF IF(7)]

restrict-top-ifex-ordered-invar[OF IF(8)], symmetric]

**note** uf1 = restrict-top-bf[OF three-ins(1) IF(2) ⟨ifex-ordered i⟩ IF(3)]

restrict-top-bf[OF three-ins(2) IF(2) ⟨ifex-ordered t⟩ IF(4)]

restrict-top-bf[OF three-ins(3) IF(2) ⟨ifex-ordered e⟩ IF(5)]

**show** ?case

**by**(rule trans[OF brace90shannon[where i=a]])

(auto simp: restrict-top-ifex-ordered-invar IF(1,2,6-8) uf1 mIH bf-ite-def[of
λl. l a]

split: ifc-split)

**qed** (simp add: bf-ite-def bf-ifex-rel-def)+

**theorem** ifex-ite-rel-bf:

(fi, i) ∈ bf-ifex-rel ⇒

(ft, t) ∈ bf-ifex-rel ⇒

(fe, e) ∈ bf-ifex-rel ⇒

((bf-ite fi ft fe), (ifex-ite i t e)) ∈ bf-ifex-rel

**by** (auto simp add: bf-ifex-rel-def order-ifex-ite-invar minimal-ifex-ite-invar val-ifex-ite
simp del: ifex-ite.simps)

**definition** *param-opt* **where** *param-opt i t e =*  
*(if i = Trueif then Some t else*  
*if i = Falseif then Some e else*  
*if t = Trueif ∧ e = Falseif then Some i else*  
*if t = e then Some t else*  
*if e = Trueif ∧ i = t then Some Trueif else*  
*if t = Falseif ∧ i = e then Some Falseif else*  
*None)*

**lemma** *param-opt-ifex-ite-eq*: *ro-ifex i ⇒ ro-ifex t ⇒ ro-ifex e ⇒*  
*param-opt i t e = Some r ⇒ r = ifex-ite i t e*  
**apply**(*rule ro-ifex-unique*)  
**subgoal** **by** (*subst (asm) param-opt-def*) (*simp split: if-split-asm*)  
**subgoal** **using** *order-ifex-ite-invar minimal-ifex-ite-invar* **by** (*blast*)  
**by** (*subst val-ifex-ite[symmetric]*)  
*(auto split: if-split-asm simp add: bf-ite-def param-opt-def val-ifex-ite[symmetric])*

**function** *ifex-ite-opt* :: *'a ifex ⇒ 'a ifex ⇒ 'a ifex ⇒ ('a :: linorder) ifex* **where**  
*ifex-ite-opt i t e = (case param-opt i t e of Some b ⇒ b | None ⇒*  
*(case lowest-tops [i, t, e] of Some x ⇒*  
*(IFC x (ifex-ite-opt (restrict-top i x True) (restrict-top t x*  
*True)*  
*(restrict-top e x True))*  
*(ifex-ite-opt (restrict-top i x False) (restrict-top t x False)*  
*(restrict-top e x False)))*  
*| None ⇒ (case i of Trueif ⇒ t | Falseif ⇒ e)))*  
**by** *pat-completeness auto*

**termination** *ifex-ite-opt*  
**by** (*relation measure (λ(i,t,e). size i + size t + size e), rule wf-measure, unfold*  
*in-measure*)  
*(simp-all only: termlemma)*

**lemma** *ifex-ite-opt-eq*:  
*ro-ifex i ⇒ ro-ifex t ⇒ ro-ifex e ⇒ ifex-ite-opt i t e = ifex-ite i t e*  
**apply**(*induction i t e rule: ifex-ite-opt.induct*)  
**apply**(*subst ifex-ite-opt.simps*)  
**apply**(*rename-tac i t e*)  
**apply**(*case-tac ∃ r. param-opt i t e = Some r*)  
**subgoal**  
**apply**(*simp del: ifex-ite.simps restrict-top.simps lowest-tops.simps*)  
**apply**(*rule param-opt-ifex-ite-eq*)  
**by** (*auto simp add: bf-ifex-rel-def*)  
**subgoal** **for** *i t e*  
**apply**(*clarsimp simp del: restrict-top.simps ifex-ite.simps ifex-ite-opt.simps*)  
**apply**(*cases lowest-tops [i,t,e] = None*)  
**subgoal** **by** *clarsimp*  
**subgoal**

```

apply(clarsimp simp del: restrict-top.simps ifex-ite.simps ifex-ite-opt.simps)
apply(subst ifex-ite.simps)
apply(rename-tac y)
apply(subgoal-tac (ifex-ite-opt (restrict-top i y True) (restrict-top t y True)
(restrict-top e y True)) =
      (ifex-ite (restrict-top i y True) (restrict-top t y True) (restrict-top
e y True)))
apply(subgoal-tac (ifex-ite-opt (restrict-top i y False) (restrict-top t y False)
(restrict-top e y False)) =
      (ifex-ite (restrict-top i y False) (restrict-top t y False) (restrict-top
e y False)))
subgoal by force
subgoal using restrict-top-ifex-minimal-invar restrict-top-ifex-ordered-invar
by metis
subgoal using restrict-top-ifex-minimal-invar restrict-top-ifex-ordered-invar
by metis
done
done
done

```

**lemma** *ro-ifexI*:  $(a,b) \in \text{bf-ifex-rel} \implies \text{ro-ifex } b$  **by** (*simp add: ifex-minimal-implied ifex-ordered-implied*)

**theorem** *ifex-ite-opt-rel-bf*:

```

(fi, i)  $\in$  bf-ifex-rel  $\implies$ 
(ft, t)  $\in$  bf-ifex-rel  $\implies$ 
(fe, e)  $\in$  bf-ifex-rel  $\implies$ 
((bf-ite fi ft fe), (ifex-ite-opt i t e))  $\in$  bf-ifex-rel
using ifex-ite-rel-bf ifex-ite-opt-eq ro-ifexI by metis

```

**lemma** *restrict-top-bf-ifex-rel*:

```

(f, i)  $\in$  bf-ifex-rel  $\implies \exists f'. (f', \text{restrict-top } i \text{ var } \text{val}) \in \text{bf-ifex-rel}$ 
unfolding bf-ifex-rel-def using restrict-top-ifex-minimal-invar restrict-top-ifex-ordered-invar
by fast

```

**lemma** *param-opt-lowest-tops-lem*:  $\text{param-opt } i \text{ t } e = \text{None} \implies \exists y. \text{lowest-tops}$   
 $[i, t, e] = \text{Some } y$

**by** (*cases i*) (*auto simp add: param-opt-def*)

**fun** *ifex-sat* **where**

```

ifex-sat Trueif = Some (const False) |
ifex-sat Falseif = None |
ifex-sat (IF v t e) =
  (case ifex-sat e of
   Some a  $\Rightarrow$  Some (a(v:=False))) |
   None  $\Rightarrow$  (case ifex-sat t of
    Some a  $\Rightarrow$  Some (a(v:=True))) |

```

*None*  $\Rightarrow$  *None*))

**lemma** *ifex-sat-untouched-False*:  $v \notin \text{ifex-var-set } i \implies \text{ifex-sat } i = \text{Some } a \implies a = \text{False}$

**proof**(*induction i arbitrary: a*)

**case** (*IF v1 t e*)

**have** *ni*:  $v \notin \text{ifex-var-set } t \ v \notin \text{ifex-var-set } e$  **using** *IF.prem*s(1) **by** *simp-all*

**have** *ne*:  $v1 \neq v$  **using** *IF.prem*s(1) **by** *force*

**show** *?case proof*(*cases ifex-sat e*)

**case** (*Some as*)

**with** *IF.prem*s(2) **have** *au*:  $a = \text{as}(v1 := \text{False})$  **by** *simp*

**moreover from** *IF.IH*(2)[*OF ni*(2)] **have**  $as = \text{False}$  **using** *Some* .

**ultimately show** *?thesis using ne by simp*

**next**

**case** *None*

**obtain** *as where Some*:  $\text{ifex-sat } t = \text{Some } as$  **using** *None IF.prem*s(2) **by** *fastforce*

**with** *IF.prem*s(2) *None* **have** *au*:  $a = \text{as}(v1 := \text{True})$  **by**(*simp*)

**moreover from** *IF.IH*(1)[*OF ni*(1)] **have**  $as = \text{False}$  **using** *Some* .

**ultimately show** *?thesis using ne by simp*

**qed**

**qed**(*simp-all add: fun-eq-iff*)

**lemma** *ifex-upd-other*:  $v \notin \text{ifex-var-set } i \implies \text{val-ifex } i (a(v:=\text{any})) = \text{val-ifex } i a$

**proof**(*induction i*)

**case** (*IF v1 t e*)

**have** *prems*:  $v \notin \text{ifex-var-set } t \ v \notin \text{ifex-var-set } e$  **using** *IF.prem*s **by** *simp-all*

**from** *IF.prem*s **have** *ne*:  $v1 \neq v$  **by** *clarsimp*

**show** *?case by*(*simp only: val-ifex.simps fun-upd-other*[*OF ne*] *ifex-vars.simps IF.IH*(1)[*OF prems*(1)] *IF.IH*(2)[*OF prems*(2)] *split: if-splits*)

**qed** *simp-all*

**fun** *ifex-no-twice where*

*ifex-no-twice* (*IF v t e*) = (

$v \notin (\text{ifex-var-set } t \cup \text{ifex-var-set } e) \wedge$   
 $\text{ifex-no-twice } t \wedge \text{ifex-no-twice } e) \mid$

*ifex-no-twice* - = *True*

**lemma** *ordered-ifex-no-twiceI*:  $\text{ifex-ordered } i \implies \text{ifex-no-twice } i$

**by**(*induction i*) (*simp-all,blast*)

**lemma** *ifex-sat-NoneD*:  $\text{ifex-sat } i = \text{None} \implies \text{val-ifex } i \text{ ass} = \text{False}$

**by**(*induction i*) (*simp-all split: option.splits*)

**lemma** *ifex-sat-SomeD*:  $\text{ifex-no-twice } i \implies \text{ifex-sat } i = \text{Some } \text{ass} \implies \text{val-ifex } i \text{ ass} = \text{True}$

**proof**(*induction i arbitrary: ass*)

**case** (*IF v t e*)

**have** *ni*:  $v \notin \text{ifex-var-set } t \ v \notin \text{ifex-var-set } e$  **using** *IF.prem*s(1) **by** *simp-all*



```

note IF.prems[unfolded ifex-sat.simps]
thus ?case proof(cases ifex-sat e)
  case (Some a) thus ?thesis using IF.prems
  apply(clarsimp simp only: val-ifex.simps ifex-sat.simps option.simps fun-upd-same
if-False ifex-upd-other[OF ni(2)])
  apply(rule IF.IH(2), simp-all)
  done
next
  case None
  obtain a where Some: ifex-sat t = Some a using None IF.prems(2) by
fastforce
  thus ?thesis using IF.prems
  by(clarsimp simp only: val-ifex.simps ifex-sat.simps option.simps fun-upd-same
if-True None ifex-upd-other[OF ni(1)])
  (rule IF.IH(1), simp-all)
  qed
qed simp-all
lemma ifex-sat-NoneI: ifex-no-twice i  $\implies$  ( $\bigwedge$  ass. val-ifex i ass = False)  $\implies$ 
ifex-sat i = None

```

```

proof(rule ccontr, goal-cases)
  case 1
  from 1(3) obtain as where ifex-sat i = Some as by blast
  from ifex-sat-SomeD[OF 1(1) this] show False using 1(2) by simp
qed

```

```

fun ifex-sat-list where
ifex-sat-list Trueif = Some [] |
ifex-sat-list Falseif = None |
ifex-sat-list (IF v t e) =
  (case ifex-sat-list e of
    Some a  $\Rightarrow$  Some ((v,False)#a) |
    None  $\Rightarrow$  (case ifex-sat-list t of
      Some a  $\Rightarrow$  Some ((v,True)#a) |
      None  $\Rightarrow$  None)))

```

**definition** *update-assignment-alt u as = ( $\lambda v.$  case map-of u v of None  $\Rightarrow$  as v |*  
*Some n  $\Rightarrow$  n)*

```

fun update-assignment where
update-assignment ((v,u)#us) as = (update-assignment us as)(v:=u) |
update-assignment [] as = as

```

**lemma** *update-assignment-notin: a  $\notin$  fst ' set us  $\implies$  update-assignment us as a*  
*= as a*

**by**(*induction us*) *clarsimp+*

**lemma** *update-assignment-alt: update-assignment u as = update-assignment-alt u*  
*as*

**by**(*induction u arbitrary: as*) (*clarsimp simp: update-assignment-alt-def fun-eq-iff*)+

**lemma** *update-assignment: distinct (map fst ((v,u)#us))  $\implies$  update-assignment ((v,u)#us) as = update-assignment us (as(v:=u))*

**unfolding** *update-assignment-alt update-assignment-alt-def*

**unfolding** *fun-eq-iff*

**by**(*clarsimp split: option.splits*) *force*

**lemma** *ass-upd-same: update-assignment ((v, u) # a) ass v = u by simp*

**lemma** *ifex-sat-list-subset: ifex-sat-list t = Some u  $\implies$  fst ' set u  $\subseteq$  ifex-var-set t*

**proof**(*induction t arbitrary: u*)

**case** (*IF v t e*)

**show** *?case*

**proof**(*cases ifex-sat-list e*)

**case** (*Some ue*)

**note** *IF.IH(2)[OF this]*

**hence** *fst ' set ue  $\subseteq$  ifex-var-set (IF v t e) by simp blast*

**moreover have** *fst ' set u = insert v (fst ' set ue) using IF.prem Some by force*

**ultimately show** *?thesis by simp*

**next**

**case** *None*

**with** *IF.prem obtain ut where Some: ifex-sat-list t = Some ut by (simp split: option.splits)*

**note** *IF.IH(1)[OF this]*

**hence** *fst ' set ut  $\subseteq$  ifex-var-set (IF v t e) by simp blast*

**moreover have** *fst ' set u = insert v (fst ' set ut) using IF.prem None Some by force*

**ultimately show** *?thesis by simp*

**qed**

**qed** *simp-all*

**lemma** *sat-list-distinct: ifex-no-twice t  $\implies$  ifex-sat-list t = Some u  $\implies$  distinct (map fst u)*

**proof**(*induction t arbitrary: u*)

**case** (*IF v t e*)

**from** *IF.prem have nt: ifex-no-twice t ifex-no-twice e by simp-all*

**note** *mIH = IF.IH(1)[OF this(1)] IF.IH(2)[OF this(2)]*

**show** *?case*

**proof**(*cases ifex-sat-list e*)

**case** (*Some a*)

**note** *mIH = mIH(2)[OF this]*

**thus** *?thesis using IF.prem ifex-sat-list.simps Some ifex-sat-list-subset by fastforce*

**next**

**case** *None*

**with** *IF.prem obtain ut where Some: ifex-sat-list t = Some ut by (simp*

*split: option.splits*  
**note** *mIH(1)[OF this]*  
**thus** *?thesis using IF.premis ifex-sat-list.simps None Some ifex-sat-list-subset*  
**by** *fastforce*  
**qed**  
**qed** *simp-all*

**lemma** *ifex-sat-list-NoneD: ifex-sat-list i = None  $\implies$  val-ifex i ass = False*

**by** *(induction i) (simp-all split: option.splits)*

**lemma** *ifex-sat-list-SomeD: ifex-no-twice i  $\implies$  ifex-sat-list i = Some u  $\implies$  ass = update-assignment u ass'  $\implies$  val-ifex i ass = True*

**proof** *(induction i arbitrary: ass ass' u)*

**case** *(IF v t e)*

**have** *nt: ifex-no-twice t ifex-no-twice e using IF.premis(1) by simp-all*

**have** *ni: v  $\notin$  ifex-var-set t v  $\notin$  ifex-var-set e using IF.premis(1) by simp-all*

**note** *IF.premis[unfolded ifex-sat.simps]*

**thus** *?case proof(cases ifex-sat-list e)*

**case** *(Some a)*

**have** *ef: u = (v, False) # a using IF.premis(2) Some by simp*

**from** *IF.premis(3) have au: ass = update-assignment a (ass'(v := False)) unfolding ef using update-assignment[OF sat-list-distinct[OF IF.premis(1,2), unfolded ef]] by presburger*

**have** *avF: ass v = False using IF.premis(3)[symmetric] unfolding ef by clarsimp*

**show** *?thesis using IF.IH(2)[OF nt(2) Some au] Some IF.premis(2) avF by simp*

**next**

**case** *None*

**obtain** *a where Some: ifex-sat-list t = Some a using None IF.premis(2) by fastforce*

**have** *ef: u = (v, True) # a using IF.premis(2) None Some by simp*

**from** *IF.premis(3) have au: ass = update-assignment a (ass'(v := True)) unfolding ef using update-assignment[OF sat-list-distinct[OF IF.premis(1,2), unfolded ef]] by presburger*

**have** *avT: ass v = True using IF.premis(3)[symmetric] unfolding ef by clarsimp*

**show** *?thesis using IF.IH(1)[OF nt(1) Some au] Some IF.premis(2) avT by simp*

**qed**

**qed** *simp-all*

**fun** *sat-list-to-bdt where*

*sat-list-to-bdt [] = Trueif |*

*sat-list-to-bdt ((v,u)#us) = (if u then IF v (sat-list-to-bdt us) Falseif else IF v Falseif (sat-list-to-bdt us))*

**lemma** *ifex-sat-list i = Some u  $\implies$  val-ifex (sat-list-to-bdt u) as  $\implies$  val-ifex i as*

**proof** *(induction i arbitrary: u)*

**case** *(IF v t e)*

```

show ?case proof(cases ifex-sat-list e)
  case (Some a)
    note mIH = IF.IH(2)[OF this]
    have ef: u = (v, False) # a using IF.prem(1) Some by simp
    have avF: as v = False using IF.prem(2) unfolding ef by(simp split:
if-splits)
    have val-ifex (sat-list-to-bdt a) as using IF.prem(2) unfolding ef using avF
by simp
    note mIH = mIH[OF this]
    thus ?thesis using avF by simp
  next
    case None
    obtain a where Some: ifex-sat-list t = Some a using None IF.prem(1) by
fastforce
    have ef: u = (v, True) # a using IF.prem(1) Some None by simp
    have avT: as v = True using IF.prem(2) unfolding ef by(simp split:
if-splits)
    have val-ifex (sat-list-to-bdt a) as using IF.prem(2) unfolding ef using avT
by simp
    note mIH = IF.IH(1)[OF Some this]
    thus ?thesis using avT by simp
  qed
qed simp-all

```

```

lemma bf-ifex-rel-consts[simp,intro!]:
  (bf-True, Trueif) ∈ bf-ifex-rel
  (bf-False, Falseif) ∈ bf-ifex-rel
by(fastforce simp add: bf-ifex-rel-def)+
lemma bf-ifex-rel-lit[simp,intro!]:
  (bf-lit v, IFC v Trueif Falseif) ∈ bf-ifex-rel
by(simp add: bf-ifex-rel-def IFC-def bf-lit-def)

```

```

lemma bf-ifex-rel-consts-ensured[simp]:
  (bf-True,x) ∈ bf-ifex-rel ↔ (x = Trueif)
  (bf-False,x) ∈ bf-ifex-rel ↔ (x = Falseif)
by(auto simp add: bf-ifex-rel-def
  intro: roifex-Trueif-unique roifex-Falseif-unique)

```

```

lemma bf-ifex-rel-consts-ensured-rev[simp]:
  (x,Trueif) ∈ bf-ifex-rel ↔ (x = bf-True)
  (x,Falseif) ∈ bf-ifex-rel ↔ (x = bf-False)
by(simp-all add: bf-ifex-rel-def fun-eq-iff)

```

```

declare ifex-ite-opt.simps restrict-top.simps lowest-tops.simps[simp del]

```

```

end

```

## 4 Option Helpers

These definitions were contributed by Peter Lammich.

```

theory Option-Helpers
imports Main HOL-Library.Monad-Syntax
begin

primrec oassert :: bool  $\Rightarrow$  unit option where
  oassert True = Some () | oassert False = None

```

```

lemma oassert-iff[simp]:
  oassert  $\Phi$  = Some x  $\longleftrightarrow$   $\Phi$ 
  oassert  $\Phi$  = None  $\longleftrightarrow$   $\neg\Phi$ 
by (cases  $\Phi$ ) auto

```

The idea is that we want the result of some computation to be *Some v* and the contents of *v* to satisfy some property *Q*.

```

primrec ospec :: ('a option)  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool where
  ospec None = False
| ospec (Some v) Q = Q v

```

**named-theorems** ospec-rules

```

lemma oreturn-rule[ospec-rules]:  $\llbracket P r \rrbracket \Longrightarrow$  ospec (Some r) P by simp

```

```

lemma obind-rule[ospec-rules]:  $\llbracket$  ospec m Q;  $\bigwedge r. Q r \Longrightarrow$  ospec (f r) P  $\rrbracket \Longrightarrow$ 
  ospec (m  $\gg$  f) P
apply (cases m)
apply (auto split: Option.bind-splits)
done

```

```

lemma ospec-alt: ospec m P = (case m of None  $\Rightarrow$  False | Some x  $\Rightarrow$  P x)
by (auto split: option.splits)

```

```

lemma ospec-bind-simp: ospec (m  $\gg$  f) P  $\longleftrightarrow$  (ospec m ( $\lambda r. ospec$  (f r) P))
apply (cases m)
apply (auto split: Option.bind-splits)
done

```

```

lemma ospec-cons:
assumes ospec m Q
assumes  $\bigwedge r. Q r \Longrightarrow$  P r
shows ospec m P
using assms by (cases m) auto

```

```

lemma oreturn-synth: ospec (Some x) ( $\lambda r. r=x$ ) by simp

```

```

lemma ospecD: ospec x P  $\Longrightarrow$  x = Some y  $\Longrightarrow$  P y by simp

```

```

lemma ospecD2: ospec x P  $\Longrightarrow$   $\exists y. x = \text{Some } y \wedge P y$  by(cases x) simp-all

```

end

## 5 Abstract ITE Implementation

```

theory Abstract-Impl
imports BDT
          Automatic-Refinement.Refine-Lib
          Option-Helpers
begin

datatype ('a, 'ni) IFEXD = TD | FD | IFD 'a 'ni 'ni

locale bdd-impl-pre =
  fixes R :: 's  $\Rightarrow$  ('ni  $\times$  ('a :: linorder) ifex) set
  fixes I :: 's  $\Rightarrow$  bool
begin
  definition les:: 's  $\Rightarrow$  's  $\Rightarrow$  bool where
    les s s' ==  $\forall$  ni n. (ni, n)  $\in$  R s  $\longrightarrow$  (ni, n)  $\in$  R s'
end

locale bdd-impl = bdd-impl-pre for R :: 's  $\Rightarrow$  ('ni  $\times$  ('a :: linorder) ifex) set +
  fixes Timpl :: 's  $\rightarrow$  ('ni  $\times$  's)
  fixes Fimpl :: 's  $\rightarrow$  ('ni  $\times$  's)
  fixes IFimpl :: 'a  $\Rightarrow$  'ni  $\Rightarrow$  'ni  $\Rightarrow$  's  $\rightarrow$  ('ni  $\times$  's)
  fixes DESTRimpl :: 'ni  $\Rightarrow$  's  $\rightarrow$  ('a, 'ni) IFEXD

  assumes Timpl-rule: I s  $\Longrightarrow$  ospec (Timpl s) ( $\lambda$ (ni, s'). (ni, Trueif)  $\in$  R s'  $\wedge$ 
I s'  $\wedge$  les s s')
  assumes Fimpl-rule: I s  $\Longrightarrow$  ospec (Fimpl s) ( $\lambda$ (ni, s'). (ni, Falseif)  $\in$  R s'  $\wedge$ 
I s'  $\wedge$  les s s')
  assumes IFimpl-rule: [I s; (ni1, n1)  $\in$  R s; (ni2, n2)  $\in$  R s]
     $\Longrightarrow$  ospec (IFimpl v ni1 ni2 s) ( $\lambda$ (ni, s'). (ni, IFC v n1 n2)  $\in$ 
R s'  $\wedge$  I s'  $\wedge$  les s s')

  assumes DESTRimpl-rule1: I s  $\Longrightarrow$  (ni, Trueif)  $\in$  R s  $\Longrightarrow$  ospec (DESTRimpl
ni s) ( $\lambda$ r. r = TD)
  assumes DESTRimpl-rule2: I s  $\Longrightarrow$  (ni, Falseif)  $\in$  R s  $\Longrightarrow$  ospec (DESTRimpl
ni s) ( $\lambda$ r. r = FD)
  assumes DESTRimpl-rule3: I s  $\Longrightarrow$  (ni, IF v n1 n2)  $\in$  R s  $\Longrightarrow$ 
    ospec (DESTRimpl ni s)
    ( $\lambda$ r.  $\exists$  ni1 ni2. r = (IFD v ni1 ni2)  $\wedge$  (ni1, n1)  $\in$  R s
 $\wedge$  (ni2, n2)  $\in$  R s)
begin

lemma les-refl[simp,intro!]:les s s by (auto simp add: les-def)
lemma les-trans[trans]:les s1 s2  $\Longrightarrow$  les s2 s3  $\Longrightarrow$  les s1 s3 by (auto simp add: les-def)
lemmas DESTRimpl-rules = DESTRimpl-rule1 DESTRimpl-rule2 DESTRimpl-rule3

```

**lemma** *DESTRimpl-rule-useless*:

$I\ s \Longrightarrow (ni, n) \in R\ s \Longrightarrow \text{ospec } (\text{DESTRimpl } ni\ s) (\lambda r. (\text{case } r \text{ of}$   
 $\quad TD \Rightarrow (ni, \text{Trueif}) \in R\ s \mid$   
 $\quad FD \Rightarrow (ni, \text{Falseif}) \in R\ s \mid$   
 $\quad IFD\ v\ nt\ ne \Rightarrow (\exists t\ e. n = IF\ v\ t\ e \wedge (ni, IF\ v\ t\ e) \in R\ s)))$

**by**(*cases n; clarify; drule (1) DESTRimpl-rules; drule ospecD2; clarsimp*)

**lemma** *DESTRimpl-rule*:

$I\ s \Longrightarrow (ni, n) \in R\ s \Longrightarrow \text{ospec } (\text{DESTRimpl } ni\ s) (\lambda r. (\text{case } n \text{ of}$   
 $\quad \text{Trueif} \Rightarrow r = TD \mid$   
 $\quad \text{Falseif} \Rightarrow r = FD \mid$   
 $\quad IF\ v\ t\ e \Rightarrow (\exists tn\ en. r = IFD\ v\ tn\ en \wedge (tn, t) \in R\ s \wedge (en, e) \in R\ s)))$

**by**(*cases n; clarify; drule (1) DESTRimpl-rules; drule ospecD2; clarsimp*)

**definition** *case-ifexi fti ffi fii ni s*  $\equiv$  *do* {

*dest*  $\leftarrow$  *DESTRimpl ni s*;  
*case dest of*  
 $\quad TD \Rightarrow fti\ s$   
 $\mid FD \Rightarrow ffi\ s$   
 $\mid IFD\ v\ ti\ ei \Rightarrow fii\ v\ ti\ ei\ s$ }

**lemma** *case-ifexi-rule*:

**assumes** *INV*:  $I\ s$

**assumes** *NI*:  $(ni, n) \in R\ s$

**assumes** *FII*:  $\llbracket n = \text{Trueif} \rrbracket \Longrightarrow \text{ospec } (fti\ s) (\lambda(r, s'). (r, ft) \in Q\ s \wedge I'\ s')$

**assumes** *FFI*:  $\llbracket n = \text{Falseif} \rrbracket \Longrightarrow \text{ospec } (ffi\ s) (\lambda(r, s'). (r, ff) \in Q\ s \wedge I'\ s')$

**assumes** *FII*:  $\bigwedge ti\ ei\ v\ t\ e. \llbracket n = IF\ v\ t\ e; (ti, t) \in R\ s; (ei, e) \in R\ s \rrbracket \Longrightarrow \text{ospec } (fii\ v\ ti\ ei\ s) (\lambda(r, s'). (r, fi\ v\ t\ e) \in Q\ s \wedge I'\ s')$

**shows**  $\text{ospec } (\text{case-ifexi } fti\ ffi\ fii\ ni\ s) (\lambda(r, s'). (r, \text{case-ifex } ft\ ff\ fi\ n) \in Q\ s \wedge I'\ s')$

**unfolding** *case-ifexi-def*

**apply** (*cases n*)

**subgoal**

**apply** (*rule obind-rule*)

**apply** (*rule DESTRimpl-rule1[OF INV]*)

**using** *NI FII* **by** (*auto*)

**subgoal**

**apply** (*rule obind-rule*)

**apply** (*rule DESTRimpl-rule2[OF INV]*)

**using** *NI FFI* **by** (*auto*)

**subgoal**

**apply** (*rule obind-rule*)

**apply** (*rule DESTRimpl-rule3[OF INV]*)

**using** *NI FII* **by** (*auto*)

**done**

**abbreviation** *return x*  $\equiv$   $\lambda s. \text{Some } (x, s)$

**primrec** *lowest-tops-impl* **where**

```

lowest-tops-impl [] s = Some (None,s) |
lowest-tops-impl (e#es) s =
  case-ifexi
    (λs. lowest-tops-impl es s)
    (λs. lowest-tops-impl es s)
    (λv t e s. do {
      (rec,s) ← lowest-tops-impl es s;
      (case rec of
        Some u ⇒ Some ((Some (min u v)), s) |
        None ⇒ Some ((Some v), s))
      }) e s

```

**declare** *lowest-tops-impl.simps*[simp del]

```

fun lowest-tops-alt where
lowest-tops-alt [] = None |
lowest-tops-alt (e#es) = (
  let rec = lowest-tops-alt es in
  case-ifex
    rec
    rec
    (λv t e. (case rec of
      Some u ⇒ (Some (min u v)) |
      None ⇒ (Some v))
    ) e
)

```

**lemma** *lowest-tops-alt*: *lowest-tops l = lowest-tops-alt l*  
**by** (*induction l rule: lowest-tops.induct*) (*auto split: option.splits simp: lowest-tops.simps*)

```

lemma lowest-tops-impl-R:
assumes list-all2 (in-rel (R s)) li l I s
shows ospec (lowest-tops-impl li s) (λ(r,s'). r = lowest-tops l ∧ s'=s)
unfolding lowest-tops-alt
using assms apply (induction rule: list-all2-induct)
subgoal by (simp add: lowest-tops-impl.simps)
subgoal
  apply (simp add: lowest-tops-impl.simps)
  apply (rule case-ifexi-rule[where Q=λs. Id, unfolded pair-in-Id-conv])
  apply assumption+
  apply (rule obind-rule)
  apply assumption
  apply (clarsimp split: option.splits)
done

```

**definition** *restrict-top-impl* **where**  
*restrict-top-impl e vr vl s =*



```

case-ifexi
  (return e)
  (return e)
  ( $\lambda v te ee. \text{return (if } v = vr \text{ then (if } vl \text{ then } te \text{ else } ee) \text{ else } e)$ )
e s

```

**lemma** *restrict-top-alt*:  $\text{restrict-top } n \text{ var } val = (\text{case } n \text{ of } (IF \ v \ t \ e) \Rightarrow (\text{if } v = \text{var then (if } val \text{ then } t \text{ else } e) \text{ else } (IF \ v \ t \ e)) \mid - \Rightarrow n)$   
**apply** (*induction*  $n \text{ var } val$  *rule*: *restrict-top.induct*)  
**apply** (*simp-all*)  
**done**

**lemma** *restrict-top-impl-spec*:  $I \ s \Longrightarrow (ni, n) \in R \ s \Longrightarrow \text{ospec (restrict-top-impl } ni \ vr \ vl \ s) (\lambda(res, s'). (res, \text{restrict-top } n \ vr \ vl) \in R \ s \wedge s' = s)$   
**unfolding** *restrict-top-impl-def restrict-top-alt*  
**by** (*rule case-ifexi-rule*[**where**  $I' = \lambda s'. s' = s$  **and**  $Q = R$ , *simplified*]) *auto*

**partial-function**(*option*) *ite-impl where*

```

ite-impl i t e s = do {
  (lt, -)  $\leftarrow$  lowest-tops-impl [i, t, e] s;
  (case lt of
    Some a  $\Rightarrow$  do {
      (ti, -)  $\leftarrow$  restrict-top-impl i a True s;
      (tt, -)  $\leftarrow$  restrict-top-impl t a True s;
      (te, -)  $\leftarrow$  restrict-top-impl e a True s;
      (fi, -)  $\leftarrow$  restrict-top-impl i a False s;
      (ft, -)  $\leftarrow$  restrict-top-impl t a False s;
      (fe, -)  $\leftarrow$  restrict-top-impl e a False s;
      (tb, s)  $\leftarrow$  ite-impl ti tt te s;
      (fb, s)  $\leftarrow$  ite-impl fi ft fe s;
      IFimpl a tb fb s}
    | None  $\Rightarrow$  case-ifexi ( $\lambda. (\text{Some } (t, s))$ ) ( $\lambda. (\text{Some } (e, s))$ ) ( $\lambda. \dots$  . None) i s
  )}

```

**lemma** *ite-impl-R*:  $I \ s \Longrightarrow \text{in-rel } (R \ s) \ ii \ i \Longrightarrow \text{in-rel } (R \ s) \ ti \ t \Longrightarrow \text{in-rel } (R \ s) \ ei \ e \Longrightarrow \text{ospec (ite-impl } ii \ ti \ ei \ s) (\lambda(r, s'). (r, \text{ifex-ite } i \ t \ e) \in R \ s' \wedge I \ s' \wedge \text{les } s \ s')$

**proof**(*induction*  $i \ t \ e$  *arbitrary*:  $s \ ii \ ti \ ei$  *rule*: *ifex-ite.induct, goal-cases*)

**case** ( $1 \ i \ t \ e \ s \ ii \ ti \ ei$ ) **note** *goal1 = 1*

**have** *la2*: *list-all2* ( $\text{in-rel } (R \ s)$ ) [ $ii, ti, ei$ ] [ $i, t, e$ ] **using** *goal1(4-6)* **by** *simp*

**show** *?case proof*(*cases lowest-tops* [ $i, t, e$ ])

**case** *None* **from** *goal1(3-6)* **show** *?thesis*

**apply**(*subst ite-impl.simps*)

**apply**(*rule obind-rule*[**where**  $Q = \lambda(r, s'). r = \text{lowest-tops } [i, t, e] \wedge s' = s$ ])

**apply**(*rule ospec-cons*)

```

    apply(rule lowest-tops-impl-R[OF la2])
    apply(assumption)
    apply(clarsimp split: prod.splits)
    apply(simp add: None split: prod.splits)
    apply(clarsimp)
    apply(rule ospec-cons)
    apply(rule case-ifexi-rule[where I'=λs'. s'=s])
    using None by (auto split: prod.splits ifex.splits simp: lowest-tops.simps)
next
case (Some lv)
note mIH = goal1(1,2)[OF Some]
from goal1(3-6) show ?thesis
  apply(subst ite-impl.simps)
  apply(rule obind-rule[where Q=λ(r, s'). r = lowest-tops [i,t,e]])
  apply(rule ospec-cons)
  apply(rule lowest-tops-impl-R[OF la2])
  apply(assumption)
  apply(clarsimp split: prod.splits)
  apply(simp add: Some split: prod.splits)
  apply(clarsimp)

  apply(rule obind-rule, rule restrict-top-impl-spec, assumption+, clarsimp
split: prod.splits)+
  apply(rule obind-rule)
  apply(rule mIH(1))
  apply(simp;fail)+
  apply(clarsimp)
  apply(rule obind-rule)
  apply(rule mIH(2))
  apply(simp add: les-def;fail)+
  apply(simp split: prod.splits)
  apply(rule ospec-cons)
  apply(rule Fimpl-rule)
  apply(simp add: les-def;fail)+
  using les-def les-trans by blast+
qed
qed

lemma case-ifexi-mono[partial-function-mono]:
  assumes [partial-function-mono]:
    mono-option (λF. fti F s)
    mono-option (λF. ffi F s)
  ∧ x31 x32 x33. mono-option (λF. fii F x31 x32 x33 s)
  shows mono-option (λF. case-ifexi (fti F) (ffi F) (fii F) ni s)
  unfolding case-ifexi-def by (tactic (Partial-Function.mono-tac @ {context} 1))

partial-function(option) val-impl :: 'ni ⇒ ('a ⇒ bool) ⇒ 's ⇒ (bool × 's) option
where
  val-impl e ass s = case-ifexi

```

```

( $\lambda s. \text{Some } (\text{True}, s)$ )
( $\lambda s. \text{Some } (\text{False}, s)$ )
( $\lambda v t e s. \text{val-impl } (\text{if } \text{ass } v \text{ then } t \text{ else } e) \text{ ass } s$ )
e s

```

**lemma**  $I s \implies (ni, n) \in R s \implies \text{ospec } (\text{val-impl } ni \text{ ass } s) (\lambda(r, s'). r = (\text{val-ifex } n \text{ ass}) \wedge s' = s)$

```

apply (induction n arbitrary: ni)
subgoal
apply (subst val-impl.simps)
apply (rule ospec-cons)
apply (rule case-ifexi-rule[where  $I' = \lambda s'. s' = s$  and  $Q = \lambda s. \text{Id}$ ]; assumption?)
by auto
subgoal
apply (subst val-impl.simps)
apply (rule ospec-cons)
apply (rule case-ifexi-rule[where  $I' = \lambda s'. s' = s$  and  $Q = \lambda s. \text{Id}$ ]; assumption?)
by auto
subgoal
apply (subst val-impl.simps)
apply (subst val-ifex.simps)
apply (clarsimp; intro impI conjI)
apply (rule ospec-cons)
apply (rule case-ifexi-rule[where  $I' = \lambda s'. s' = s$  and  $Q = \lambda s. \text{Id}$ ]; assumption?)
apply (simp; fail)
apply (simp; fail)
apply (rule ospec-cons)
apply (rprems; simp; fail)
apply (simp; fail)
apply (simp; fail)
apply (rule ospec-cons)
apply (rule case-ifexi-rule[where  $I' = \lambda s'. s' = s$  and  $Q = \lambda s. \text{Id}$ ]; assumption?)
apply (simp; fail)
apply (simp; fail)
apply (simp)
apply (rule ospec-cons)
apply (rprems; simp; fail)
apply (simp; fail)
apply (simp; fail)
done
done

```

**end**

**locale** *bdd-impl-cmp-pre* = *bdd-impl-pre*  
**begin**

**definition** *map-invar-impl* m s =  
 $(\forall ii ti ei ri. m (ii, ti, ei) = \text{Some } ri \implies$

$(\exists i t e. ((ri, ifex-ite-opt i t e) \in R s) \wedge (ii, i) \in R s \wedge (ti, t) \in R s \wedge (ei, e) \in R s))$

**lemma** *map-invar-impl-les*:  $map-invar-impl m s \implies les s s' \implies map-invar-impl m s'$

**unfolding** *map-invar-impl-def bdd-impl-pre.les-def* **by** *blast*

**lemma** *map-invar-impl-update*:  $map-invar-impl m s \implies$

$(ii, i) \in R s \implies (ti, t) \in R s \implies (ei, e) \in R s \implies$

$(ri, ifex-ite-opt i t e) \in R s \implies map-invar-impl (m((ii, ti, ei) \mapsto ri)) s$

**unfolding** *map-invar-impl-def* **by** *auto*

**end**

**locale** *bdd-impl-cmp* = *bdd-impl* + *bdd-impl-cmp-pre* +

**fixes**  $M :: 'a \Rightarrow ('b \times 'b \times 'b) \Rightarrow 'b \text{ option}$

**fixes**  $U :: 'a \Rightarrow ('b \times 'b \times 'b) \Rightarrow 'b \Rightarrow 'a$

**fixes**  $cmp :: 'b \Rightarrow 'b \Rightarrow \text{bool}$

**assumes** *cmp-rule1*:  $I s \implies (ni, i) \in R s \implies (ni', i) \in R s \implies cmp ni ni'$

**assumes** *cmp-rule2*:  $I s \implies cmp ni ni' \implies (ni, i) \in R s \implies (ni', i') \in R s \implies i = i'$

**assumes** *map-invar-rule1*:  $I s \implies map-invar-impl (M s) s$

**assumes** *map-invar-rule2*:  $I s \implies (ii, it) \in R s \implies (ti, tt) \in R s \implies (ei, et) \in R s \implies$

$(ri, ifex-ite-opt it tt et) \in R s \implies U s (ii, ti, ei) ri = s' \implies I s'$

**assumes** *map-invar-rule3*:  $I s \implies R (U s (ii, ti, ei) ri) = R s$

**begin**

**lemma** *cmp-rule-eq*:  $I s \implies (ni, i) \in R s \implies (ni', i') \in R s \implies cmp ni ni' \longleftrightarrow i = i'$

**using** *cmp-rule1 cmp-rule2* **by** *force*

**lemma** *DESTRimpl-Some*:  $I s \implies (ni, i) \in R s \implies ospec (DESTRimpl ni s) (\lambda r. True)$

**apply**(*cases i*)

**apply**(*auto intro: ospec-cons dest: DESTRimpl-rules*)

**done**

**fun** *param-opt-impl* **where**

*param-opt-impl i t e s* = **do** {

*ii*  $\leftarrow$  *DESTRimpl i s*;

*ti*  $\leftarrow$  *DESTRimpl t s*;

*ei*  $\leftarrow$  *DESTRimpl e s*;

(*tn, s*)  $\leftarrow$  *Timpl s*;

(*fn, s*)  $\leftarrow$  *Fimpl s*;

*Some* ((*if ii = TD then Some t else*

*if ii = FD then Some e else*

*if ti = TD  $\wedge$  ei = FD then Some i else*

```

    if cmp t e then Some t else
    if ei = TD ∧ cmp i t then Some tn else
    if ti = FD ∧ cmp i e then Some fn else
    None), s)}

```

**declare** *param-opt-impl.simps*[simp del]

**lemma** *param-opt-impl-lesI*:

```

assumes I s (ii,i) ∈ R s (ti,t) ∈ R s (ei,e) ∈ R s
shows ospec (param-opt-impl ii ti ei s)
           (λ(r,s'). I s' ∧ les s s')
using assms apply(subst param-opt-impl.simps)
by (auto simp add: param-opt-def les-def intro!: obind-rule
      dest: DESTRimpl-Some Timpl-rule Fimpl-rule)

```

**lemma** *param-opt-impl-R*:

```

assumes I s (ii,i) ∈ R s (ti,t) ∈ R s (ei,e) ∈ R s
shows ospec (param-opt-impl ii ti ei s)
           (λ(r,s'). case r of None ⇒ param-opt i t e = None
                    | Some r ⇒ (∃ r'. param-opt i t e = Some r' ∧ (r, r')
                    ∈ R s'))
using assms apply(subst param-opt-impl.simps)
apply(rule obind-rule)
apply(rule DESTRimpl-rule; assumption)
apply(rule obind-rule)
apply(rule DESTRimpl-rule; assumption)
apply(rule obind-rule)
apply(rule DESTRimpl-rule; assumption)
apply(rule obind-rule)
apply(rule Timpl-rule; assumption)
apply(safe)
apply(rule obind-rule)
apply(rule Fimpl-rule; assumption)
by (auto simp add: param-opt-def les-def cmp-rule-eq split: ifex.splits)

```

**partial-function**(option) *ite-impl-opt* **where**

```

ite-impl-opt i t e s = do {
  (ld, s) ← param-opt-impl i t e s;
  (case ld of Some b ⇒ Some (b, s) |
   None ⇒
  do {
    (lt,-) ← lowest-tops-impl [i, t, e] s;
    (case lt of
     Some a ⇒ do {
       (ti,-) ← restrict-top-impl i a True s;
       (tt,-) ← restrict-top-impl t a True s;
       (te,-) ← restrict-top-impl e a True s;
       (fi,-) ← restrict-top-impl i a False s;
       (ft,-) ← restrict-top-impl t a False s;

```

```

    (fe,-) ← restrict-top-impl e a False s;
    (tb,s) ← ite-impl-opt ti tt te s;
    (fb,s) ← ite-impl-opt fi ft fe s;
    IFimpl a tb fb s}
  | None ⇒ case-ifexi (λ-.(Some (t,s))) (λ-.(Some (e,s))) (λ- - - . None) i s
)}))}

```

**lemma** *ospec-and*:  $ospec\ f\ P \implies ospec\ f\ Q \implies ospec\ f\ (\lambda x. P\ x \wedge Q\ x)$   
**using** *ospecD2* **by** *force*

**lemma** *ite-impl-opt-R*:

```

  I s
  ⇒ in-rel (R s) ii i ⇒ in-rel (R s) ti t ⇒ in-rel (R s) ei e
  ⇒ ospec (ite-impl-opt ii ti ei s) (λ(r, s'). (r, ifex-ite-opt i t e) ∈ R s' ∧ I s' ∧
les s s')

```

**proof**(*induction i t e arbitrary: s ii ti ei rule: ifex-ite-opt.induct, goal-cases*)

**note** *ifex-ite-opt.simps*[*simp del*] *restrict-top.simps*[*simp del*]

**case** (1 i t e s ii ti ei) **note** *goal1* = 1

**have** *la2*: *list-all2* (in-rel (R s)) [ii,ti,ei] [i,t,e] **using** *goal1*(4-6) **by** *simp*

**note** *mIH* = *goal1*(1,2)

**from** *goal1*(3-6) **show** ?*case*

**apply**(*cases param-opt i t e*)

**defer**

**apply**(*subst ite-impl-opt.simps*)

**apply**(*rule obind-rule*)

**apply**(*rule ospec-and*[*OF param-opt-impl-R param-opt-impl-lesI*])

**apply**(*auto simp add: les-def ifex-ite-opt.simps split: option.splits*)[9]

**apply**(*frule param-opt-lowest-tops-lem*)

**apply**(*clarsimp*)

**apply**(*subst ite-impl-opt.simps*)

**apply**(*rule obind-rule*)

**apply**(*rule ospec-and*[*OF param-opt-impl-R param-opt-impl-lesI*])

**apply**(*auto split: option.splits*)[8]

**apply**(*clarsimp split: option.splits*)

**apply**(*rule obind-rule*[**where**  $Q = \lambda(r, s'). r = \text{lowest-tops } [i, t, e]$ ])

**apply**(*rule ospec-cons*)

**apply**(*rule lowest-tops-impl-R*)

**using** *les-def* **apply**(*fastforce*)

**apply**(*assumption*)

**apply**(*fastforce*)

**using** *BDT.param-opt-lowest-tops-lem* **apply**(*clarsimp split: prod.splits*)

**apply**(*rule obind-rule, rule restrict-top-impl-spec, assumption, auto simp add:*  
*les-def split: prod.splits*)+

**apply**(*rule obind-rule*)

**apply**(*rule mIH*(1))

**apply**(*simp add: les-def:fail*)+

**apply**(*clarsimp*)

```

    apply(rule obind-rule)
    apply(rule mIH(2))
      apply(simp add: les-def:fail)+
    apply(simp add: ifex-ite-opt.simps split: prod.splits)
    apply(rule ospec-cons)
    apply(rule FImpl-rule)
      apply(auto simp add: les-def:fail)+
    done
qed

```

```

partial-function (option) ite-impl-lu where
ite-impl-lu i t e s = do {
  (case M s (i,t,e) of Some b  $\Rightarrow$  Some (b,s) | None  $\Rightarrow$  do {
    (ld, s)  $\leftarrow$  param-opt-impl i t e s;
    (case ld of Some b  $\Rightarrow$  Some (b, s) |
    None  $\Rightarrow$ 
    do {
      (lt,-)  $\leftarrow$  lowest-tops-impl [i, t, e] s;
      (case lt of
      Some a  $\Rightarrow$  do {
        (ti,-)  $\leftarrow$  restrict-top-impl i a True s;
        (tt,-)  $\leftarrow$  restrict-top-impl t a True s;
        (te,-)  $\leftarrow$  restrict-top-impl e a True s;
        (fi,-)  $\leftarrow$  restrict-top-impl i a False s;
        (ft,-)  $\leftarrow$  restrict-top-impl t a False s;
        (fe,-)  $\leftarrow$  restrict-top-impl e a False s;
        (tb,s)  $\leftarrow$  ite-impl-lu ti tt te s;
        (fb,s)  $\leftarrow$  ite-impl-lu fi ft fe s;
        (r,s)  $\leftarrow$  FImpl a tb fb s;
        let s = U s (i,t,e) r;
        Some (r,s)
      } |
      None  $\Rightarrow$  None
    )
  )
})}}}}

```

```

declare ifex-ite-opt.simps[simp del]

```

```

lemma ite-impl-lu-R: I s
 $\implies$  (ii,i)  $\in$  R s  $\implies$  (ti,t)  $\in$  R s  $\implies$  (ei,e)  $\in$  R s
 $\implies$  ospec (ite-impl-lu ii ti ei s)
  ( $\lambda$ (r, s'). (r, ifex-ite-opt i t e)  $\in$  R s'  $\wedge$  I s'  $\wedge$  les s s')

```

```

proof(induction i t e arbitrary: s ii ti ei rule: ifex-ite-opt.induct, goal-cases)
note restrict-top.simps[simp del]
case (1 i t e s ii ti ei) note goal1 = 1
have la2: list-all2 (in-rel (R s)) [ii,ti,ei] [i,t,e] using goal1(4-6) by simp
note mIH = goal1(1,2)
from goal1(3-6) show ?case
  apply(subst ite-impl-lu.simps)
  apply(cases M s (ii, ti, ei))

```

**defer**

```
apply(frul map-invar-rule1)
apply(simp only: option.simps ospec.simps prod.simps simp-thms les-refl)
apply(subst (asm) map-invar-impl-def)
apply(erule allE[where x = ii])
apply(erule allE[where x = ti])
apply(erule allE[where x = ei])
apply(rename-tac a)
apply(erule-tac x = a in allE)
apply(metis cmp-rule-eq)
```

```
apply(clarsimp)
apply(cases param-opt i t e)
defer
```

```
apply(rule obind-rule)
  apply(rule ospec-and[OF param-opt-impl-R param-opt-impl-lesI])
    apply(auto simp add: map-invar-impl-les ifex-ite-opt.simps split:
option.splits)[9]
```

```
apply(frul param-opt-lowest-tops-lem)
apply(clarsimp)
apply(rule obind-rule)
  apply(rule ospec-and[OF param-opt-impl-R param-opt-impl-lesI])
    apply(auto split: option.splits)[8]
  apply(clarsimp split: option.splits)
  apply(rule-tac obind-rule[where Q= $\lambda(r, s^{\wedge}). r = \text{lowest-tops } [i, t, e]$ ])
  apply(rule ospec-cons)
    apply(rule lowest-tops-impl-R)
      using les-def apply(fastforce)
    apply(assumption)
  apply(fastforce)
  using BDT.param-opt-lowest-tops-lem apply(clarsimp split: prod.splits)
  apply(rule obind-rule, rule restrict-top-impl-spec, assumption+, auto simp add:
les-def split: prod.splits)+
  apply(rule obind-rule)
  apply(rule mIH(1))
    apply(simp add: map-invar-impl-les les-def;fail)+
  apply(clarsimp)
  apply(rule obind-rule)
  apply(rule mIH(2))
    apply(simp add: map-invar-impl-les les-def;fail)+
  apply(simp add: ifex-ite-opt.simps split: prod.splits)
  apply(rule obind-rule)
  apply(rule Fimpl-rule)
    apply(simp)
  apply(auto simp add: les-def)[2]
  apply(clarsimp simp add: les-def)
```



```

apply(safe)
using map-invar-rule3 apply(presburger)
apply(rule map-invar-rule2)
  prefer 6 apply(blast)
  apply(blast)
  apply(blast)
  apply(blast)
  apply(blast)
apply(clarsimp simp add: ifex-ite-opt.simps)
using map-invar-rule3 by presburger
qed

end
end

```

## 6 Pointermap

```

theory Pointer-Map
imports Main
begin

```

We need a datastructure that supports the following two operations:

- Given an element, it can construct a pointer (i.e., a small representation) of that element. It will always construct the same pointer for equal elements.
- Given a pointer, we can retrieve the element

```

record 'a pointermap =
  entries :: 'a list
  getentry :: 'a ⇒ nat option

```

```

definition pointermap-sane m ≡ (distinct (entries m) ∧
  (∀ n ∈ {.. $\text{length}$  (entries m)}. getentry m (entries m ! n) = Some n) ∧
  (∀ p i. getentry m p = Some i → entries m ! i = p ∧ i < length (entries m)))

```

```

definition empty-pointermap ≡ (|entries = [], getentry = λp. None |)

```

```

lemma pointermap-empty-sane[simp, intro!]: pointermap-sane empty-pointermap

```

```

unfolding empty-pointermap-def pointermap-sane-def by simp

```

```

definition pointermap-insert a m ≡ (|entries = (entries m)@[a], getentry = (getentry m)(a ↦ length (entries m))|)

```

```

definition pm-pth m p ≡ entries m ! p

```

```

definition pointermap-p-valid p m ≡ p < length (entries m)

```

**definition** *pointermap-getmk*  $a\ m \equiv (\text{case } \text{getentry } m\ a \text{ of } \text{Some } p \Rightarrow (p, m) \mid \text{None} \Rightarrow \text{let } u = \text{pointermap-insert } a\ m \text{ in } (\text{the } (\text{getentry } u\ a), u))$

**lemma** *pointermap-sane-appendD*:  $\text{pointermap-sane } s \Longrightarrow m \notin \text{set } (\text{entries } s) \Longrightarrow \text{pointermap-sane } (\text{pointermap-insert } m\ s)$

**unfolding** *pointermap-sane-def pointermap-insert-def*

**proof**(*intro conjI[rotated], goal-cases*)

**case 3 thus ?case by simp**

**next**

**case 2**

  {

**fix**  $n$

**have**  $\llbracket \text{distinct } (\text{entries } s) \wedge (\forall x. x \in \{..<\text{length } (\text{entries } s)\} \longrightarrow \text{getentry } s (\text{entries } s ! x) = \text{Some } x) \wedge (\forall p\ i. \text{getentry } s\ p = \text{Some } i \longrightarrow \text{entries } s ! i = p \wedge i < \text{length } (\text{entries } s)); m \notin \text{set } (\text{entries } s);$

$n \in \{..<\text{length } (\text{entries } (\text{entries } = \text{entries } s @ [m], \text{getentry } = \text{getentry } s(m \mapsto \text{length } (\text{entries } s))))\}; n < \text{length } (\text{entries } s)\llbracket$

$\Longrightarrow \text{getentry } (\text{entries } = \text{entries } s @ [m], \text{getentry } = \text{getentry } s(m \mapsto \text{length } (\text{entries } s))) (\text{entries } (\text{entries } = \text{entries } s @ [m], \text{getentry } = \text{getentry } s(m \mapsto \text{length } (\text{entries } s))) ! n) = \text{Some } n$

$\llbracket \text{distinct } (\text{entries } s) \wedge (\forall x. x \in \{..<\text{length } (\text{entries } s)\} \longrightarrow \text{getentry } s (\text{entries } s ! x) = \text{Some } x) \wedge (\forall p\ i. \text{getentry } s\ p = \text{Some } i \longrightarrow \text{entries } s ! i = p \wedge i < \text{length } (\text{entries } s)); m \notin \text{set } (\text{entries } s);$

$n \in \{..<\text{length } (\text{entries } (\text{entries } = \text{entries } s @ [m], \text{getentry } = \text{getentry } s(m \mapsto \text{length } (\text{entries } s))))\}; \neg n < \text{length } (\text{entries } s)\llbracket$

$\Longrightarrow \text{getentry } (\text{entries } = \text{entries } s @ [m], \text{getentry } = \text{getentry } s(m \mapsto \text{length } (\text{entries } s))) (\text{entries } (\text{entries } = \text{entries } s @ [m], \text{getentry } = \text{getentry } s(m \mapsto \text{length } (\text{entries } s))) ! n) = \text{Some } n$

**proof**(*goal-cases*)

**case 1 note**  $\text{goal1} = 1$

**from**  $\text{goal1}(4)$  **have**  $sa: \bigwedge a. (\text{entries } s @ a) ! n = \text{entries } s ! n$  **by** (*simp add: nth-append*)

**from**  $\text{goal1}(1,4)$  **have**  $ih: \text{getentry } s (\text{entries } s ! n) = \text{Some } n$  **by** *simp*

**from**  $\text{goal1}(2,4)$  **have**  $ne: \text{entries } s ! n \neq m$  **using** *nth-mem* **by** *fastforce*

**from**  $sa\ ih\ ne$  **show** *?case* **by** *simp*

**next**

**case 2 note**  $\text{goal2} = 2$

**from**  $\text{goal2}(3,4)$  **have**  $ln: n = \text{length } (\text{entries } s)$  **by** *simp*

**hence**  $sa: \bigwedge a. (\text{entries } s @ [a]) ! n = a$  **by** *simp*

**from**  $sa\ ln$  **show** *?case* **by** *simp*

**qed**

  } **note**  $h = \text{this}$

**with 2 show** *?case* **by** *blast*

**next**

**case 1 thus ?case**

**by**(*clarsimp simp add: nth-append fun-upd-same Ball-def*) *force*

**qed**

**lemma** *lentries-noneD*:  $getentry\ s\ a = None \implies pointermap\ sane\ s \implies a \notin set\ (entries\ s)$   
**unfolding** *pointermap-sane-def*  
**proof**(*rule ccontr, goal-cases*)  
  **case** 1  
  **from** 1(3) **obtain**  $n$  **where**  $n < length\ (entries\ s)$   $entries\ s\ !\ n = a$  **unfolding** *in-set-conv-nth* **by** *blast*  
  **with** 1(2,1) **show** *False* **by** *force*  
**qed**

**lemma** *pm-pth-append*:  $pointermap\ p\ valid\ p\ m \implies pm\ pth\ (pointermap\ insert\ a\ m)\ p = pm\ pth\ m\ p$   
**unfolding** *pointermap-p-valid-def pm-pth-def pointermap-insert-def*  
**by**(*simp add: nth-append*)

**lemma** *pointermap-insert-in*:  $u = (pointermap\ insert\ a\ m) \implies pm\ pth\ u\ (the\ (getentry\ u\ a)) = a$   
**unfolding** *pointermap-insert-def pm-pth-def*  
**by**(*simp*)

**lemma** *pointermap-insert-p-validI*:  $pointermap\ p\ valid\ p\ m \implies pointermap\ p\ valid\ p\ (pointermap\ insert\ a\ m)$   
**unfolding** *pointermap-insert-def pointermap-p-valid-def*  
**by** *simp*

**thm** *nth-eq-iff-index-eq*  
**lemma** *pth-eq-iff-index-eq*:  $pointermap\ sane\ m \implies pointermap\ p\ valid\ p1\ m \implies pointermap\ p\ valid\ p2\ m \implies (pm\ pth\ m\ p1 = pm\ pth\ m\ p2) \longleftrightarrow (p1 = p2)$   
**unfolding** *pointermap-sane-def pointermap-p-valid-def pm-pth-def*  
**using** *nth-eq-iff-index-eq* **by** *blast*

**lemma** *pointermap-p-valid-updateI*:  $pointermap\ sane\ m \implies getentry\ m\ a = None \implies u = pointermap\ insert\ a\ m \implies p = the\ (getentry\ u\ a) \implies pointermap\ p\ valid\ p\ u$   
**by**(*simp add: pointermap-sane-def pointermap-p-valid-def pointermap-insert-def*)

**lemma** *pointermap-get-validI*:  $pointermap\ sane\ m \implies getentry\ m\ a = Some\ p \implies pointermap\ p\ valid\ p\ m$   
**by**(*simp add: pointermap-sane-def pointermap-p-valid-def*)

**lemma** *pointermap-sane-getmkD*:  
  **assumes** *sn: pointermap-sane m*  
  **assumes** *res: pointermap-getmk a m = (p,u)*  
  **shows**  $pointermap\ sane\ u \wedge pointermap\ p\ valid\ p\ u$   
**using** *sn res[symmetric]*  
  **apply**(*cases getentry m a*)  
  **apply**(*simp-all add: pointermap-getmk-def Let-def split: option.split*)  
  **apply**(*rule*)  
  **apply**(*rule pointermap-sane-appendD*)

```

    apply (clarify; fail)+
    apply (rule luentries-noneD)
    apply (clarify; fail)+
    apply (rule pointermap-p-valid-updateI [OF - - refl refl])
    apply (clarify; fail)+
    apply (erule pointermap-get-validI)
  by simp

lemma pointermap-update-pthI:
  assumes sn: pointermap-sane m
  assumes res: pointermap-getmk a m = (p,u)
  shows pm-pth u p = a
using assms
  apply (simp add: pointermap-getmk-def Let-def split: option.splits)
  apply (meson pointermap-insert-in)
  apply (clarsimp simp: pointermap-sane-def pm-pth-def)
done

lemma pointermap-p-valid-inv:
  assumes pointermap-p-valid p m
  assumes pointermap-getmk a m = (x,u)
  shows pointermap-p-valid p u
using assms
  by (simp add: pointermap-getmk-def Let-def split: option.splits) (meson pointermap-insert-p-validI)

lemma pointermap-p-pth-inv:
  assumes pv: pointermap-p-valid p m
  assumes u: pointermap-getmk a m = (x,u)
  shows pm-pth u p = pm-pth m p
using pm-pth-append [OF pv] u
  by (clarsimp simp: pointermap-getmk-def Let-def split: option.splits)

lemma pointermap-backward-valid:
  assumes puw: pointermap-p-valid p u
  assumes u: pointermap-getmk a m = (x,u)
  assumes ne: x ≠ p
  shows pointermap-p-valid p m

using assms
  by (auto simp: Let-def pointermap-getmk-def pointermap-p-valid-def pointermap-insert-def
  split: option.splits)

end

```

## 7 Functional interpretation for the abstract implementation

```
theory Middle-Impl
```

```
imports Abstract-Impl Pointer-Map
begin
```

For the lack of a better name, the suffix *mi* stands for middle-implementation. This reflects that this “implementation” is neither entirely abstract, nor has it been made fully concrete: the data structures are decided, but not their implementations.

```
record bdd =
  dpm :: (nat × nat × nat) pointermap
  dcl :: ((nat × nat × nat), nat) map
```

```
definition emptymi ≡ (|dpm = empty-pointermap, dcl = Map.empty|)
```

```
fun destrmi :: nat ⇒ bdd ⇒ (nat, nat) IFEXD where
destrmi 0 bdd = FD |
destrmi (Suc 0) bdd = TD |
destrmi (Suc (Suc n)) bdd = (case pm-pth (dpm bdd) n of (v, t, e) ⇒ IFD v t e)
fun tmi where tmi bdd = (1, bdd)
fun fmi where fmi bdd = (0, bdd)
fun ifmi :: nat ⇒ nat ⇒ nat ⇒ bdd ⇒ (nat × bdd) where
ifmi v t e bdd = (if t = e
  then (t, bdd)
  else (let (r, pm) = pointermap-getmk (v, t, e) (dpm bdd) in
    (Suc (Suc r), dpm-update (const pm) bdd)))
```

```
fun Rmi-g :: nat ⇒ nat ifex ⇒ bdd ⇒ bool where
Rmi-g 0 Falseif bdd = True |
Rmi-g (Suc 0) Trueif bdd = True |
Rmi-g (Suc (Suc n)) (IF v t e) bdd = (pointermap-p-valid n (dpm bdd)
  ∧ (case pm-pth (dpm bdd) n of (nv, nt, ne) ⇒ nv = v ∧ Rmi-g nt t bdd ∧ Rmi-g ne e bdd)) |
Rmi-g - - - = False
```

```
definition Rmi s ≡ {(a,b)|a b. Rmi-g a b s}
```

```
interpretation mi-pre: bdd-impl-cmp-pre Rmi by -
```

```
definition bdd-node-valid bdd n ≡ n ∈ Domain (Rmi bdd)
```

```
lemma [simp]:
  bdd-node-valid bdd 0
  bdd-node-valid bdd (Suc 0)
  apply (simp-all add: bdd-node-valid-def Rmi-def)
  using Rmi-g.simps(1,2) apply blast+
done
```

```
definition ifexd-valid bdd e ≡ (case e of IFD - t e ⇒ bdd-node-valid bdd t ∧
bdd-node-valid bdd e | - ⇒ True)
```

```
definition bdd-sane bdd ≡ pointermap-sane (dpm bdd) ∧ mi-pre.map-invar-impl
```

(*dcl bdd*) *bdd*

**lemma** [*simp,intro!*]: *bdd-sane emptymi*  
**unfolding** *emptymi-def bdd-sane-def bdd.simps*  
**by**(*simp add: mi-pre.map-invar-impl-def*)

**lemma** *prod-split3*:  $P (\text{case } p \text{ of } (x, xa, xaa) \Rightarrow f x xa xaa) = (\forall x1 x2 x3. p = (x1, x2, x3) \longrightarrow P (f x1 x2 x3))$   
**by**(*simp split: prod.splits*)

**lemma** *IfI*:  $(c \Longrightarrow P x) \Longrightarrow (\neg c \Longrightarrow P y) \Longrightarrow P (\text{if } c \text{ then } x \text{ else } y)$  **by** *simp*

**lemma** *fstsndI*:  $x = (a,b) \Longrightarrow \text{fst } x = a \wedge \text{snd } x = b$  **by** *simp*

**thm** *nat.split*

**lemma** *Rmi-g-2-split*:  $P (Rmi-g \ n \ x \ m) = ((x = \text{Falseif} \longrightarrow P (Rmi-g \ n \ x \ m)) \wedge (x = \text{Trueif} \longrightarrow P (Rmi-g \ n \ x \ m)) \wedge (\forall \text{vs ts es. } x = \text{IF vs ts es} \longrightarrow P (Rmi-g \ n \ x \ m)))$   
**by**(*cases x;simp*)

**lemma** *rmigeq*:  $Rmi-g \ ni1 \ n1 \ s \Longrightarrow Rmi-g \ ni2 \ n2 \ s \Longrightarrow ni1 = ni2 \Longrightarrow n1 = n2$

**proof**(*induction ni1 n1 s arbitrary: n2 ni2 rule: Rmi-g.induct, goal-cases*)

**case** ( $3 \ n \ v \ t \ e \ bdd \ n2 \ ni2$ ) **note** *goal3 = 3*

**note**  $1 = \text{goal3}(1,2)$

**have**  $2$ :  $Rmi-g \ (\text{fst} \ (\text{snd} \ (\text{pm-pth} \ (\text{dpm} \ bdd) \ n))) \ t \ bdd \ Rmi-g \ (\text{snd} \ (\text{snd} \ (\text{pm-pth} \ (\text{dpm} \ bdd) \ n))) \ e \ bdd$  **using** *goal3(3)* **by**(*clarsimp*)+

**note** *mIH* =  $1(1)[\text{OF} \ - \ - \ 2(1) \ - \ \text{refl}] \ 1(2)[\text{OF} \ - \ - \ 2(2) \ - \ \text{refl}]$

**obtain**  $v2 \ t2 \ e2$  **where**  $v2: n2 = \text{IF } v2 \ t2 \ e2$  **using** *Rmi-g.simps(4,6)* *goal3(3-5)*  
**by**(*cases n2*) *blast+*

**thus** *?case* **using** *goal3(3-4)* **by**(*clarsimp simp add: v2 goal3(5)[symmetric]*)  
*mIH*)

**qed** (*rename-tac n2 ni2, (case-tac n2; clarsimp)*)+

**lemma** *rmigneq*:  $bdd\text{-sane } s \Longrightarrow Rmi-g \ ni1 \ n1 \ s \Longrightarrow Rmi-g \ ni2 \ n2 \ s \Longrightarrow ni1 \neq ni2 \Longrightarrow n1 \neq n2$

**proof**(*induction ni1 n1 s arbitrary: n2 ni2 rule: Rmi-g.induct, goal-cases*)

**case**  $1$  **thus** *?case* **by** (*metis Rmi-g.simps(6) old.nat.exhaust*)

**next**

**case**  $2$  **thus** *?case* **by** (*metis Rmi-g.simps(4,8) old.nat.exhaust*)

**next**

**case** ( $3 \ n \ v \ t \ e \ bdd \ n2 \ ni2$ ) **note** *goal3 = 3*

**let** *?bddpth* = *pm-pth (dpm bdd)*

**note**  $1 = \text{goal3}(1,2)[\text{OF } \text{prod.collapse } \text{prod.collapse}]$

**have**  $2$ :  $Rmi-g \ (\text{fst} \ (\text{snd} \ (\text{?bddpth} \ n))) \ t \ bdd \ Rmi-g \ (\text{snd} \ (\text{snd} \ (\text{?bddpth} \ n))) \ e \ bdd$  **using** *goal3(4)* **by**(*clarsimp*)+

**note** *mIH* =  $1(1)[\text{OF } \text{goal3}(3) \ 2(1)] \ 1(2)[\text{OF } \text{goal3}(3) \ 2(2)]$

**show** *?case* **proof**(*cases 0 < ni2, case-tac 1 < ni2*)

**case** *False*

**hence**  $e: ni2 = 0$  **by** *simp*

**with** *goal3(5)* **have**  $n2 = \text{Falseif}$  **using** *rmigeq* **by** *auto*

```

    thus ?thesis by simp
  next
    case True moreover assume  $\exists: \neg 1 < ni2$ 
    ultimately have  $ni2 = 1$  by simp
    with goal3(5) have  $n2 = Trueif$  using rmigeq by auto
    thus ?thesis by simp
  next
    assume  $\exists: 1 < ni2$ 
    then obtain  $ni2s$  where [simp]:  $ni2 = Suc (Suc ni2s)$  unfolding One-nat-def
  using less-imp-Suc-add by blast
    obtain  $v2 t2 e2$  where  $v2[simp]: n2 = IF v2 t2 e2$  using goal3(5) by (cases
    ( $ni2, n2, bdd$ ) rule: Rmi-g.cases) clarsimp+
    have  $4: Rmi-g (fst (snd (?bddpth ni2s))) t2 bdd Rmi-g (snd (snd (?bddpth
    ni2s))) e2 bdd$  using goal3(5) by clarsimp+
    show ?case unfolding  $v2$ 
    proof (cases  $fst (snd (?bddpth n)) = fst (snd (?bddpth ni2s))$ ,
      case-tac  $snd (snd (?bddpth n)) = snd (snd (?bddpth ni2s))$ ,
      case-tac  $v = v2$ )
      have  $ne: ni2s \neq n$  using goal3(6) by simp
      have  $ib: pointermap-p-valid n (dpm bdd) pointermap-p-valid ni2s (dpm bdd)$ 
    using Rmi-g.simps(3) goal3(4,5) by simp-all
      assume goal1:
         $fst (snd (pm-pth (dpm bdd) n)) = fst (snd (pm-pth (dpm bdd) ni2s))$ 
         $snd (snd (pm-pth (dpm bdd) n)) = snd (snd (pm-pth (dpm bdd) ni2s))$ 
         $v = v2$ 
      hence  $?bddpth n = ?bddpth ni2s$  unfolding prod-eq-iff using goal3(4)
    goal3(5) by auto
      with goal3(3)  $ne$  have False unfolding bdd-sane-def using pth-eq-iff-index-eq[OF
    -  $ib$ ] by simp
      thus  $IF v t e \neq IF v2 t2 e2 ..$ 
    qed (simp-all add:  $mIH(1)[OF 4(1)] mIH(2)[OF 4(2)]$ )
  qed
qed simp-all

```

```

lemma ifmi-les-hlp:  $pointermap-sane (dpm s) \implies pointermap-getmk (v, ni1, ni2)$ 
 $(dpm s) = (x1, dpm s') \implies Rmi-g nia n s \implies Rmi-g nia n s'$ 
proof (induction  $nia n s$  rule: Rmi-g.induct, goal-cases)
  case ( $\exists n v t e bdd$ ) note goal3 = 3
  obtain  $x1a x2a$  where  $pth[simp]: pm-pth (dpm bdd) n = (v, x1a, x2a)$  using
  goal3(5) by force
  have  $pth'[simp]: pm-pth (dpm s') n = (v, x1a, x2a)$  unfolding  $pth[symmetric]$ 
  using goal3(4,5) by (meson Rmi-g.simps(3) pointermap-p-pth-inv)
  note  $mIH = goal3(1,2)[OF pth[symmetric] refl goal3(3,4)]$ 
  from goal3(5) show ?case
    unfolding Rmi-g.simps
    using pointermap-p-valid-inv[OF - goal3(4)] mIH
    by (simp split: prod.splits)
qed simp-all
lemma ifmi-les:

```

```

    assumes bdd-sane s
    assumes ifmi v ni1 ni2 s = (ni, s')
    shows mi-pre.les s s'
using assms
by (clarsimp simp: bdd-sane-def comp-def apfst-def map-prod-def mi-pre.les-def Rmi-def
    ifmi-les-hlp split: if-splits prod.splits)

lemma ifmi-notouch-dcl: ifmi v ni1 ni2 s = (ni, s')  $\implies$  dcl s' = dcl s
  by (clarsimp split: if-splits prod.splits)

lemma ifmi-saneI: bdd-sane s  $\implies$  ifmi v ni1 ni2 s = (ni, s')  $\implies$  bdd-sane s'
  apply (subst bdd-sane-def)
  apply (rule conjI)
  apply (clarsimp simp: comp-def apfst-def map-prod-def bdd-sane-def split: if-splits
    option.splits split: prod.splits)
  apply (rule conjunct1 [OF pointermap-sane-getmkD, of dpm s (v, ni1, ni2) -])
  apply (simp-all)[2]
  apply (frule (1) ifmi-les)
  apply (unfold bdd-sane-def, clarify)
  apply (rule mi-pre.map-invar-impl-les [rotated])
  apply assumption
  apply (drule ifmi-notouch-dcl)
  apply (simp)
done

lemma rmiqif: Rmi-g ni (IF v n1 n2) s  $\implies$   $\exists$  n. ni = Suc (Suc n)
  apply (cases ni)
  apply (simp split: if-splits prod.splits)
  apply (rename-tac nis)
  apply (case-tac nis)
  apply (simp split: if-splits prod.splits)
  apply (simp split: if-splits prod.splits)
done

lemma in-lesI:
  assumes mi-pre.les s s'
  assumes (ni1, n1)  $\in$  Rmi s
  assumes (ni2, n2)  $\in$  Rmi s
  shows (ni1, n1)  $\in$  Rmi s' (ni2, n2)  $\in$  Rmi s'
by (meson assms mi-pre.les-def)+

lemma ifmi-modification-validI:
  assumes sane: bdd-sane s
  assumes ifm: ifmi v ni1 ni2 s = (ni, s')
  assumes vld: bdd-node-valid s n
  shows bdd-node-valid s' n
proof (cases ni1 = ni2)
  case True with ifm vld show ?thesis by simp

```



```

next
  case False
  {
    fix b
    from ifm have  $(n, b) \in Rmi\ s \implies (n, b) \in Rmi\ s'$ 
      by(induction n b - rule: Rmi-g.induct) (auto dest: pointermap-p-pth-inv
pointermap-p-valid-inv simp: apfst-def map-prod-def False Rmi-def split: prod.splits)
    }
    thus ?thesis
    using vld unfolding bdd-node-valid-def by blast
  }
qed

```

**definition** *tmi'* *s*  $\equiv$  do {*oassert* (*bdd-sane s*); *Some* (*tmi s*)}

**definition** *fmi'* *s*  $\equiv$  do {*oassert* (*bdd-sane s*); *Some* (*fmi s*)}

**definition** *ifmi'* *v ni1 ni2 s*  $\equiv$  do {*oassert* (*bdd-sane s*  $\wedge$  *bdd-node-valid s ni1*  $\wedge$  *bdd-node-valid s ni2*); *Some* (*ifmi v ni1 ni2 s*)}

**lemma** *ifmi'-spec*:  $\llbracket bdd-sane\ s;\ bdd-node-valid\ s\ ni1;\ bdd-node-valid\ s\ ni2 \rrbracket \implies$   
*ospec* (*ifmi' v ni1 ni2 s*) ( $\lambda r. r = ifmi\ v\ ni1\ ni2\ s$ )

**unfolding** *ifmi'-def* **by** (*simp split: Option.bind-splits*)

**lemma** *ifmi'-ifmi*:  $\llbracket bdd-sane\ s;\ bdd-node-valid\ s\ ni1;\ bdd-node-valid\ s\ ni2 \rrbracket \implies$   
*ifmi' v ni1 ni2 s* = *Some* (*ifmi v ni1 ni2 s*)

**unfolding** *ifmi'-def* **by** (*simp split: Option.bind-splits*)

**definition** *destrmi'* *ni s*  $\equiv$  do {*oassert* (*bdd-sane s*  $\wedge$  *bdd-node-valid s ni*); *Some* (*destrmi ni s*)}

**lemma** *destrmi-someD*: *destrmi' e bdd* = *Some x*  $\implies$  *bdd-sane bdd*  $\wedge$  *bdd-node-valid bdd e*

**by** (*simp add: destrmi'-def split: Option.bind-splits*)

**lemma** *Rmi-sv*:

**assumes** *bdd-sane s* (*ni,n*)  $\in Rmi\ s$  (*ni',n'*)  $\in Rmi\ s$

**shows** *ni=ni'*  $\implies n=n'$

**and** *ni $\neq$ ni'*  $\implies n\neq n'$

**using** *assms*

**apply** *safe*

**apply** (*simp-all add: Rmi-def*)

**using** *rmigeq apply simp*

**apply** (*drule* (3) *rmigneq*)

**by** *clarify*

**lemma** *True-rep[simp]*: *bdd-sane s*  $\implies (ni, Trueif) \in Rmi\ s \iff ni = Suc\ 0$

**using** *Rmi-def Rmi-g.simps(2) Rmi-sv(2)* **by** *blast*

**lemma** *False-rep[simp]*: *bdd-sane s*  $\implies (ni, Falseif) \in Rmi\ s \iff ni = 0$

**using** *Rmi-def Rmi-g.simps(1) Rmi-sv(2)* **by** *blast*

**definition** *updS* *s x r* = *dcl-update* ( $\lambda m. m(x \mapsto r)$ ) *s*

```

thm Rmi-g.induct

lemma updS-dpm: dpm (updS s x r) = dpm s
  unfolding updS-def by simp

lemma updS-Rmi-g: Rmi-g n i (updS s x r) = Rmi-g n i s
  apply(induction n i s rule: Rmi-g.induct)
  apply(simp-all) unfolding updS-dpm by auto

lemma updS-Rmi: Rmi (updS s x r) = Rmi s
  unfolding Rmi-def updS-Rmi-g by blast

interpretation mi: bdd-impl-cmp bdd-sane Rmi tmi' fmi' ifmi' destrmi' dcl updS
op =
proof –
  note s = mi-pre.les-def[simp] Rmi-def

  note [simp] = tmi'-def fmi'-def ifmi'-def destrmi'-def apfst-def map-prod-def

  show bdd-impl-cmp bdd-sane Rmi tmi' fmi' ifmi' destrmi' dcl updS (op =)
  proof(unfold-locales, goal-cases)
    case 1 thus ?case by(clarsimp split: if-splits simp: Rmi-def)
    next case 2 thus ?case by(clarsimp split: if-splits simp: Rmi-def)
    next case (3 s ni1 n1 ni2 n2 v) note goal3 = 3
      note [simp] = Rmi-sv[OF this]
      have e: n1 = n2  $\implies$  ni1 = ni2 by(rule ccontr) simp
      obtain ni s' where[simp]: (ifmi' v ni1 ni2 s) = Some (ni, s')
        unfolding ifmi'-def bdd-node-valid-def using goal3 by(simp add: DomainI
del: ifmi.simps) fastforce
        hence ifm: ifmi v ni1 ni2 s = (ni, s')
          using goal3 unfolding ifmi'-def bdd-node-valid-def
          by(simp add: DomainI)
        have ifmi'-ospec:  $\bigwedge P. ospec (ifmi' v ni1 ni2 s) P \longleftrightarrow P (ifmi v ni1 ni2 s)$ 
by(simp del: ifmi'-def ifmi.simps add: ifm)
        from goal3 show ?case
          unfolding ifmi'-ospec
          apply(split prod.splits; clarify)
          apply(rule conjI)

      apply(clarsimp simp: Rmi-def IFC-def bdd-sane-def ifmi-les-hlp pointermap-sane-getmkD
pointermap-update-pthI split: if-splits prod.splits)

      using ifmi-les[OF  $\langle$ bdd-sane s $\rangle$  ifm] ifmi-saneI[OF  $\langle$ bdd-sane s $\rangle$  ifm] ifm
apply(simp)
      done
    next case 4 thus ?case
      apply (clarsimp split: Option.bind-splits if-splits)
      done

```

```

next case 5 thus ?case by(clarsimp split: if-splits)
next case 6 thus ?case
  apply (clarsimp simp add: bdd-node-valid-def split: Option.bind-splits if-splits)
  apply (auto simp: Rmi-def elim: Rmi-g.elims)
  done
next
  case 7 thus ?case using Rmi-sv by blast
next
  case 8 thus ?case using Rmi-sv by blast
next
  case 9 thus ?case unfolding bdd-sane-def by simp
next
  case 10 thus ?case unfolding bdd-sane-def mi-pre.map-invar-impl-def using
  updS-Rmi
  by(clarsimp simp add: updS-def simp del: ifex-ite-opt.simps) blast
next
  case 11 thus ?case using updS-Rmi by auto
qed
qed

```

**lemma** *p-valid-RmiI*:  $(\text{Suc } (\text{Suc } na), b) \in \text{Rmi } bdd \implies \text{pointermmap-p-valid } na$   
 $(\text{dpm } bdd)$

**unfolding** *Rmi-def* **by**(cases b) (auto)

**lemma** *n-valid-RmiI*:  $(na, b) \in \text{Rmi } bdd \implies \text{bdd-node-valid } bdd na$

**unfolding** *bdd-node-valid-def*

**by**(intro *DomainI*, *assumption*)

**lemma** *n-valid-Rmi-alt*:  $\text{bdd-node-valid } bdd na \longleftrightarrow (\exists b. (na, b) \in \text{Rmi } bdd)$

**unfolding** *bdd-node-valid-def*

**by** *auto*

**lemma** *ifmi-result-validI*:

**assumes** *sane*: *bdd-sane s*

**assumes** *vld*: *bdd-node-valid s ni1 bdd-node-valid s ni2*

**assumes** *ifm*: *ifmi v ni1 ni2 s = (ni, s')*

**shows** *bdd-node-valid s' ni*

**proof** –

**from** *vld* **obtain** *n1 n2* **where**  $(ni1, n1) \in \text{Rmi } s$   $(ni2, n2) \in \text{Rmi } s$  **unfolding**  
*bdd-node-valid-def* **by** *blast*

**note** *mi.IFimpl-rule[OF sane this]*

**note** *this[unfolded ifmi'-ifmi[OF sane vld] ospec.simps, of v, unfolded ifm, un-*  
*folded prod.simps]*

**thus** *?thesis* **unfolding** *bdd-node-valid-def* **by** *blast*

**qed**

**end**

## 8 Array List

Most of this has been contributed by Peter Lammich.

```
theory Array-List
imports
  Separation-Logic-Imperative-HOL.Array-Blit
begin
```

This implements a datastructure that efficiently supports two operations: appending an element and looking up the  $n$ th element. The implementation is straightforward.

As underlying data structure an array is used. Since changing the length of an array requires copying, we double the size whenever the array needs to be expanded. We use a counter for the current length to track which elements are used and which are spares.

```
type-synonym 'a array-list = 'a array  $\times$  nat
```

```
definition is-array-list  $l \equiv \lambda(a,n). \exists_A l'. a \mapsto_a l' * \uparrow(n \leq \text{length } l' \wedge l = \text{take } n \text{ } l' \wedge \text{length } l' > 0)$ 
```

```
definition initial-capacity  $\equiv 16::\text{nat}$ 
```

```
definition arl-empty  $\equiv \text{do } \{$ 
   $a \leftarrow \text{Array.new initial-capacity default};$ 
   $\text{return } (a,0)$ 
 $\}$ 
```

```
lemma [sep-heap-rules]:  $\langle \text{emp} \rangle \text{ arl-empty } \langle \text{is-array-list } [] \rangle$ 
by (sep-auto simp: arl-empty-def is-array-list-def initial-capacity-def)
```

```
definition arl-nth  $\equiv \lambda(a,n) i. \text{do } \{$ 
   $\text{Array.nth } a \ i$ 
 $\}$ 
```

```
lemma [sep-heap-rules]:  $i < \text{length } l \implies \langle \text{is-array-list } l \ a \rangle \text{ arl-nth } a \ i \langle \lambda x. \text{is-array-list } l \ a * \uparrow(x = l!i) \rangle$ 
by (sep-auto simp: arl-nth-def is-array-list-def split: prod.splits)
```

```
definition arl-append  $\equiv \lambda(a,n) x. \text{do } \{$ 
   $\text{len} \leftarrow \text{Array.len } a;$ 
  

   $\text{if } n < \text{len} \text{ then do } \{$ 
     $a \leftarrow \text{Array.upd } n \ x \ a;$ 
     $\text{return } (a, n+1)$ 
   $\}$   $\text{else do } \{$ 
     $\text{let newcap} = 2 * \text{len};$ 
     $a \leftarrow \text{array-grow } a \ \text{newcap} \ \text{default};$ 
   $\}$ 
 $\}$ 
```

```

    a ← Array.upd n x a;
    return (a,n+1)
  }
}

```

```

lemma [sep-heap-rules]:
  < is-array-list l a >
  arl-append a x
  < λa. is-array-list (l@[x]) a >t
  by (sep-auto
    simp: arl-append-def is-array-list-def take-update-last neq-Nil-conv
    split: prod.splits nat.split)

```

```

lemma is-array-list-prec: precise is-array-list
  unfolding is-array-list-def[abs-def]
  apply(rule preciseI)
  apply(simp split: prod.splits)
  using preciseD snga-prec by fastforce

```

```

lemma is-array-list-lengthIA: is-array-list l li  $\implies_A \uparrow(\text{snd } li = \text{length } l) * \text{true}$ 
  by(sep-auto simp: is-array-list-def split: prod.splits)
  find-consts assn  $\implies$  bool

```

```

lemma is-array-list-lengthI:  $x \models \text{is-array-list } l \text{ li} \implies \text{snd } li = \text{length } l$ 
  using is-array-list-lengthIA by (metis (full-types) ent-pure-post-iff star-aci(2))

```

end

## 9 Imperative implementation for Pointermap

```

theory Pointer-Map-Impl

```

```

imports Array-List

```

```

  Separation-Logic-Imperative-HOL.Sep-Main

```

```

  Separation-Logic-Imperative-HOL.Hash-Map-Impl

```

```

  Pointer-Map

```

```

begin

```

```

  record 'a pointermap-impl =

```

```

    entriesi :: 'a array-list

```

```

    getentryi :: ('a,nat) hashtable

```

```

  lemma pointermapieq-exhaust: entries a = entries b  $\implies$  getentry a = getentry
  b  $\implies$  a = (b :: 'a pointermap) by simp

```

```

  definition is-pointermap-impl :: ('a::{hashable,heap}) pointermap  $\implies$  'a pointermap-impl
   $\implies$  assn where

```

```

    is-pointermap-impl b bi  $\equiv$ 

```

```

    is-array-list (entries b) (entriesi bi)

```

```

    * is-hashtable (getentry b) (getentryi bi)

```

```

  lemma is-pointermap-impl-prec: precise is-pointermap-impl

```

```

  unfolding is-pointermmap-impl-def [abs-def]
apply(rule preciseI)
apply(clarsimp)
apply(rename-tac a a' x y p F F')
apply(rule pointermmapieq-exhaust)
apply(rule-tac p = entriesi p and h = (x,y) in preciseD[OF is-array-list-prec])
apply(unfold star-aci(1))
apply blast
apply(rule-tac p = getentryi p and h = (x,y) in preciseD[OF is-hashmap-prec])
apply(simp only: star-aci(2)[symmetric])
apply(simp only: star-aci(1)[symmetric])
apply(simp only: star-aci(2)[symmetric])
done

```

```

definition pointermmap-empty where
  pointermmap-empty  $\equiv$  do {
    hm  $\leftarrow$  hm-new;
    arl  $\leftarrow$  arl-empty;
    return ( $\langle$ entriesi = arl, getentryi = hm  $\rangle$ )
  }

```

```

lemma [sep-heap-rules]:  $\langle$ emp  $\rangle$  pointermmap-empty  $\langle$ is-pointermmap-impl empty-pointermmap $\rangle_t$ 
  unfolding is-pointermmap-impl-def
  by (sep-auto simp: pointermmap-empty-def empty-pointermmap-def)

```

```

definition pm-pthi where
  pm-pthi m p  $\equiv$  arl-nth (entriesi m) p

```

```

lemma [sep-heap-rules]: pointermmap-sane m  $\impl$  pointermmap-p-valid p m  $\impl$ 
   $\langle$ is-pointermmap-impl m mi  $\rangle$  pm-pthi mi p  $\langle$  $\lambda$ ai. is-pointermmap-impl m mi *
 $\uparrow$ (ai = pm-pth m p) $\rangle$ 
  by (sep-auto simp: pm-pthi-def pm-pth-def is-pointermmap-impl-def pointermmap-p-valid-def)

```

```

definition pointermmap-getmki where
  pointermmap-getmki a m  $\equiv$  do {
    lo  $\leftarrow$  ht-lookup a (getentryi m);
    (case lo of
      Some l  $\Rightarrow$  return (l,m) |
      None  $\Rightarrow$  do {
        p  $\leftarrow$  return (snd (entriesi m));
        ent  $\leftarrow$  arl-append (entriesi m) a;
        lut  $\leftarrow$  hm-update a p (getentryi m);
        u  $\leftarrow$  return ( $\langle$ entriesi = ent, getentryi = lut $\rangle$ );
        return (p,u)
      }
    )
  }

```

```

lemmas pointermmap-getmki-defs = pointermmap-getmki-def pointermmap-getmk-def

```

```

pointermap-insert-def is-pointermap-impl-def
  lemma [sep-heap-rules]: pointermap-sane m  $\implies$  pointermap-getmk a m = (p,u)
 $\implies$ 
  < is-pointermap-impl m mi >
  pointermap-getmki a mi
  < $\lambda$ (pi,ui). is-pointermap-impl u ui *  $\uparrow$ (pi = p)>t
  apply(cases getentry m a)
  apply(unfold pointermap-getmki-def)
  apply(unfold return-bind)
  apply(rule bind-rule[where R =  $\lambda$ r. is-pointermap-impl m mi *  $\uparrow$ (r = None  $\wedge$ 
(snd (entriesi mi) = p)) * true])
  apply(sep-auto simp: pointermap-getmki-defs is-array-list-def split: prod.splits;fail)
  apply(sep-auto simp: pointermap-getmki-defs)+
  done

end

```

## 10 Imperative implementation

```

theory Conc-Impl
imports Pointer-Map-Impl Middle-Impl
begin

record bddi =
  dpmi :: (nat  $\times$  nat  $\times$  nat) pointermap-impl
  dcli :: ((nat  $\times$  nat  $\times$  nat),nat) hashtable
lemma bdd-exhaust: dpm a = dpm b  $\implies$  dcl a = dcl b  $\implies$  a = (b :: bdd) by simp

instantiation prod :: (default, default) default
begin
  definition default-prod :: ('a  $\times$  'b)  $\equiv$  (default, default)
  instance ..
end

instantiation nat :: default
begin
  definition default-nat  $\equiv$  0 :: nat
  instance ..
end

definition is-bdd-impl (bdd::bdd) (bddi::bddi) = is-pointermap-impl (dpm bdd)
(dpmi bddi) * is-hashmap (dcl bdd) (dcli bddi)

lemma is-bdd-impl-prec: precise is-bdd-impl
  apply(rule preciseI)
  apply(unfold is-bdd-impl-def)
  apply(clarsimp)
  apply(rename-tac a a' x y p F F')
  apply(rule bdd-exhaust)

```

```

apply(rule-tac p = dpmi p and h = (x,y) in preciseD[OF is-pointermap-impl-prec])
apply(unfold star-aci(1))
apply blast
apply(rule-tac p = dcli p and h = (x,y) in preciseD[OF is-hashmap-prec])
apply(simp only: star-aci(2)[symmetric])
apply(simp only: star-aci(1)[symmetric])
apply(simp only: star-aci(2)[symmetric])

```

**done**

**definition** emptyci :: bddi Heap  $\equiv$  do { ep  $\leftarrow$  pointermap-empty; ehm  $\leftarrow$  hm-new; return (|dpmi=ep, dcli=ehm|) }

**definition** tci bdd  $\equiv$  return (1::nat, bdd::bddi)

**definition** fci bdd  $\equiv$  return (0::nat, bdd::bddi)

**definition** ifci v t e bdd  $\equiv$  (if t = e then return (t, bdd) else do { (p,u)  $\leftarrow$  pointermap-getmki (v, t, e) (dpmi bdd); return (Suc (Suc p), dpmi-update (const u) bdd) })

**definition** destrci :: nat  $\Rightarrow$  bddi  $\Rightarrow$  (nat, nat) IFEXD Heap **where**

destrci n bdd  $\equiv$  (case n of

0  $\Rightarrow$  return FD |

Suc 0  $\Rightarrow$  return TD |

Suc (Suc p)  $\Rightarrow$  pm-pthi (dpmi bdd) p  $\gg$  (lambda(v,t,e). return (IFD v t e)))

**term** mi.les

**lemma** emptyci-rule[sep-heap-rules]:  $\langle emp \rangle$  emptyci  $\langle is-bdd-impl emptymi \rangle_t$   
**by** (sep-auto simp: is-bdd-impl-def emptyci-def emptymi-def)

**lemma** [sep-heap-rules]: tmi' bdd = Some (p, bdd')  
 $\Rightarrow \langle is-bdd-impl bdd bddi \rangle$   
 tci bddi  
 $\langle \lambda(pi, bddi'). is-bdd-impl bdd' bddi' * \uparrow(pi = p) \rangle$   
**by** (sep-auto simp: tci-def tmi'-def split: Option.bind-splits)

**lemma** [sep-heap-rules]: fmi' bdd = Some (p, bdd')  
 $\Rightarrow \langle is-bdd-impl bdd bddi \rangle$   
 fci bddi  
 $\langle \lambda(pi, bddi'). is-bdd-impl bdd' bddi' * \uparrow(pi = p) \rangle$   
**by** (sep-auto simp: fci-def fmi'-def split: Option.bind-splits)

**lemma** [sep-heap-rules]: ifmi' v t e bdd = Some (p, bdd')  $\Rightarrow$   
 $\langle is-bdd-impl bdd bddi \rangle$  ifci v t e bddi  
 $\langle \lambda(pi, bddi'). is-bdd-impl bdd' bddi' * \uparrow(pi = p) \rangle_t$   
**apply** (clarsimp simp: is-bdd-impl-def ifmi'-def simp del: ifmi.simps)  
**by** (sep-auto simp: ifci-def apfst-def map-prod-def is-bdd-impl-def bdd-sane-def  
 split: prod.splits if-splits Option.bind-splits)

**lemma** destrci-rule[sep-heap-rules]:



$\text{destrmi}'\ n\ \text{bdd} = \text{Some } r \implies$   
 $\langle \text{is-bdd-impl } \text{bdd } \text{bddi} \rangle\ \text{destrci } n\ \text{bddi}$   
 $\langle \lambda r'. \text{is-bdd-impl } \text{bdd } \text{bddi} * \uparrow(r' = r) \rangle$   
**unfolding**  $\text{destrmi}'\text{-def}$  **apply** ( $\text{clarsimp } \text{split}: \text{Option.bind-splits}$ )  
**apply**( $\text{cases } (n, \text{bdd})\ \text{rule}: \text{destrmi.cases}$ )  
**by** ( $\text{sep-auto } \text{simp}: \text{destrci-def } \text{bdd-node-valid-def } \text{is-bdd-impl-def } \text{ifexd-valid-def}$   
 $\text{bdd-sane-def}$   
 $\text{dest}: \text{p-valid-RmiI}$ ) $+$

**term**  $\text{mi.restrict-top-impl}$

**thm**  $\text{mi.case-ifexi-def}$

**definition**  $\text{case-ifexici } \text{fti } \text{ffi } \text{fii } \text{ni } \text{bddi} \equiv \text{do } \{$   
 $\text{dest} \leftarrow \text{destrci } \text{ni } \text{bddi};$   
 $\text{case } \text{dest of } \text{TD} \Rightarrow \text{fti} \mid \text{FD} \Rightarrow \text{ffi} \mid \text{IFD } v\ \text{ti } \text{ei} \Rightarrow \text{fii } v\ \text{ti } \text{ei}$   
 $\}$

**lemma** [ $\text{sep-decon-rules}$ ]:

**assumes**  $S: \text{mi.case-ifexi } \text{fti } \text{ffi } \text{fii } \text{ni } \text{bdd} = \text{Some } r$   
**assumes** [ $\text{sep-heap-rules}$ ]:  
 $\text{destrmi}'\ \text{ni } \text{bdd} = \text{Some } \text{TD} \implies \text{fti } \text{bdd} = \text{Some } r \implies \langle \text{is-bdd-impl } \text{bdd } \text{bddi} \rangle$   
 $\text{ftci } \langle Q \rangle$   
 $\text{destrmi}'\ \text{ni } \text{bdd} = \text{Some } \text{FD} \implies \text{ffi } \text{bdd} = \text{Some } r \implies \langle \text{is-bdd-impl } \text{bdd } \text{bddi} \rangle$   
 $\text{ffci } \langle Q \rangle$   
 $\bigwedge v\ t\ e. \text{destrmi}'\ \text{ni } \text{bdd} = \text{Some } (\text{IFD } v\ t\ e) \implies \text{fii } v\ t\ e\ \text{bdd} = \text{Some } r$   
 $\implies \langle \text{is-bdd-impl } \text{bdd } \text{bddi} \rangle\ \text{fici } v\ t\ e\ \langle Q \rangle$   
**shows**  $\langle \text{is-bdd-impl } \text{bdd } \text{bddi} \rangle\ \text{case-ifexici } \text{ftci } \text{ffci } \text{fici } \text{ni } \text{bddi } \langle Q \rangle$   
**using**  $S$  **unfolding**  $\text{mi.case-ifexi-def}$  **apply** ( $\text{clarsimp } \text{split}: \text{Option.bind-splits}$   
 $\text{IFEXD.splits}$ )  
**by** ( $\text{sep-auto } \text{simp}: \text{case-ifexici-def}$ ) $+$

**definition**  $\text{restrict-topci } p\ \text{vr } \text{vl } \text{bdd} =$

$\text{case-ifexici}$   
 $(\text{return } p)$   
 $(\text{return } p)$   
 $(\lambda v\ te\ ee. \text{return } (\text{if } v = \text{vr then } (\text{if } \text{vl then } \text{te else } \text{ee}) \text{ else } p))$   
 $p\ \text{bdd}$

**lemma** [ $\text{sep-heap-rules}$ ]:

**assumes**  $\text{mi.restrict-top-impl } p\ \text{var } \text{val } \text{bdd} = \text{Some } (r, \text{bdd}')$   
**shows**  $\langle \text{is-bdd-impl } \text{bdd } \text{bddi} \rangle\ \text{restrict-topci } p\ \text{var } \text{val } \text{bddi}$   
 $\langle \lambda ri. \text{is-bdd-impl } \text{bdd } \text{bddi} * \uparrow(ri = r) \rangle$   
**using**  $\text{assms}$  **unfolding**  $\text{mi.restrict-top-impl-def } \text{restrict-topci-def}$  **by**  $\text{sep-auto}$

**fun**  $\text{lowest-topsci}$  **where**

$\text{lowest-topsci } []\ s = \text{return } \text{None} \mid$   
 $\text{lowest-topsci } (e\#\text{es})\ s =$

```

case-ifexici
  (lowest-topsci es s)
  (lowest-topsci es s)
  ( $\lambda v t e. do \{$ 
    (rec)  $\leftarrow$  lowest-topsci es s;
    (case rec of
      Some u  $\Rightarrow$  return ((Some (min u v))) |
      None  $\Rightarrow$  return ((Some v)))
  }) e s

```

**declare** *lowest-topsci.simps*[*simp del*]

**lemma** [*sep-heap-rules*]:

```

assumes mi.lowest-topsci-impl es bdd = Some (r,bdd')
shows  $\langle is\text{-}bdd\text{-}impl\ bdd\ bddi \rangle\ lowest-topsci es bddi$ 
 $\langle \lambda(ri). is\text{-}bdd\text{-}impl\ bdd\ bddi * \uparrow(ri = r \wedge bdd' = bdd) \rangle$ 

```

**proof** –

**note** [*simp*] = *lowest-topsci.simps mi.lowest-topsci-impl.simps*

**show** *?thesis using assms*

**apply** (*induction es arbitrary: bdd r bdd' bddi*)

**apply** (*sep-auto*)

**apply** (*clarsimp simp: mi.case-ifexi-def split: Option.bind-splits IFEXD.splits*)

**apply** (*sep-auto simp: mi.case-ifexi-def*)

**apply** (*sep-auto simp: mi.case-ifexi-def*)

**apply** (*sep-auto simp: mi.case-ifexi-def*)

**done**

**qed**

**partial-function**(*heap*) *iteci where*

*iteci i t e s = do* {

(*lt*)  $\leftarrow$  *lowest-topsci [i, t, e] s*;

*case lt of*

*Some a*  $\Rightarrow$  *do* {

*ti*  $\leftarrow$  *restrict-topci i a True s*;

*tt*  $\leftarrow$  *restrict-topci t a True s*;

*te*  $\leftarrow$  *restrict-topci e a True s*;

*fi*  $\leftarrow$  *restrict-topci i a False s*;

*ft*  $\leftarrow$  *restrict-topci t a False s*;

*fe*  $\leftarrow$  *restrict-topci e a False s*;

(*tb,s'*)  $\leftarrow$  *iteci ti tt te s*;

(*fb,s''*)  $\leftarrow$  *iteci fi ft fe s'*;

(*ifci a tb fb s''*)

}

| *None*  $\Rightarrow$  *do* {

*case-ifexici (return (t,s)) (return (e,s))* ( $\lambda - - . raise$  "Cannot happen") *i s*

}

}

**declare** *iteci.simps*[*code*]

```

lemma iteci-rule:
  ( mi.ite-impl i t e bdd = Some (p,bdd') )  $\longrightarrow$ 
  <is-bdd-impl bdd bddi>
  iteci i t e bddi
  < $\lambda(pi,bddi'). is-bdd-impl bdd' bddi' * \uparrow(pi=p) >_t$ >
apply (induction arbitrary: i t e bddi bdd p bdd' rule: mi.ite-impl.fixp-induct)
  subgoal
  apply simp
  using option-admissible[where P=
     $\lambda((x1,x2),x3),x4) (r1,r2). \forall bddi.$ 
    <is-bdd-impl x4 bddi>
    iteci x1 x2 x3 bddi
    < $\lambda r. case r of (pi, bddi') \Rightarrow is-bdd-impl r2 bddi' * \uparrow(pi = r1) >_t$ >]
  apply auto[1]
  apply (fo-rule subst[rotated])
  apply (assumption)
  by auto
subgoal by simp
subgoal
  apply clarify
  apply (clarsimp split: option.splits Option.bind-splits prod.splits)
  apply (subst iteci.simps)
  apply (sep-auto)
  apply (subst iteci.simps)
  apply (sep-auto)
  unfolding imp-to-meta apply rprems
  apply simp
  apply sep-auto
  apply (rule fi-rule)
  apply rprems
  apply simp
  apply frame-inference
  by sep-auto
done

```

```

declare iteci-rule[THEN mp, sep-heap-rules]

```

```

definition param-optci where
  param-optci i t e bdd = do {
    (tr, bdd)  $\leftarrow$  tci bdd;
    (fl, bdd)  $\leftarrow$  fci bdd;
    id  $\leftarrow$  destrci i bdd;
    td  $\leftarrow$  destrci t bdd;
    ed  $\leftarrow$  destrci e bdd;
    return (
      if id = TD then Some t else
      if id = FD then Some e else
      if td = TD  $\wedge$  ed = FD then Some i else
    )
  }

```

```

    if t = e then Some t else
    if ed = TD ∧ i = t then Some tr else
    if td = FD ∧ i = e then Some fl else
    None, bdd)
}

```

**lemma** *param-optci-rule*:

```

( mi.param-opt-impl i t e bdd = Some (p,bdd') ) ==>
<is-bdd-impl bdd bddi>
  param-optci i t e bddi
  <λ(pi,bddi'). is-bdd-impl bdd' bddi' * ↑(pi=p)>_t
by (sep-auto simp add: mi.param-opt-impl.simps param-optci-def tmi'-def fmi'-def
      split: Option.bind-splits)

```

**lemma** *bdd-hm-lookup-rule*:

```

(dcl bdd (i,t,e) = p) ==>
<is-bdd-impl bdd bddi>
  hm-lookup (i, t, e) (dcli bddi)
  <λ(pi). is-bdd-impl bdd bddi * ↑(pi = p)>_t
unfolding is-bdd-impl-def by (sep-auto)

```

**lemma** *bdd-hm-update-rule*'[sep-heap-rules]:

```

<is-bdd-impl bdd bddi>
  hm-update k v (dcli bddi)
  <λr. is-bdd-impl (updS bdd k v) (dcli-update (const r) bddi) * true>
unfolding is-bdd-impl-def updS-def by (sep-auto)

```

**partial-function**(heap) *iteci-lu* **where**

```

iteci-lu i t e s = do {
  lu ← ht-lookup (i,t,e) (dcli s);
  (case lu of Some b ⇒ return (b,s)
  | None ⇒ do {
    (po,s) ← param-optci i t e s;
    (case po of Some b ⇒ do {
      return (b,s)}
    | None ⇒ do {
      (lt) ← lowest-topsci [i, t, e] s;
      (case lt of Some a ⇒ do {
        ti ← restrict-topci i a True s;
        tt ← restrict-topci t a True s;
        te ← restrict-topci e a True s;
        fi ← restrict-topci i a False s;
        ft ← restrict-topci t a False s;
        fe ← restrict-topci e a False s;
        (tb,s) ← iteci-lu ti tt te s;
        (fb,s) ← iteci-lu fi ft fe s;
        (r,s) ← ifci a tb fb s;
        cl ← hm-update (i,t,e) r (dcli s);
        return (r,dcli-update (const cl) s)
      }
    }
  }
}

```

```

    }
  | None  $\Rightarrow$  raise "Cannot happen" })
}}

```

**term** *ht-lookup*

**declare** *iteci-lu.simps*[code]

**thm** *iteci-lu.simps*[unfolded restrict-topci-def case-ifexici-def param-optci-def lowest-topsci.simps]

**partial-function**(*heap*) *iteci-lu-code* **where** *iteci-lu-code* *i t e s* = do {

*lu*  $\leftarrow$  *hm-lookup* (*i*, *t*, *e*) (*dcli s*);

*case lu of* None  $\Rightarrow$  *let po = if i = 1 then Some t*

*else if i = 0 then Some e else if t = 1  $\wedge$  e = 0 then Some*

*i else if t = e then Some t else if e = 1  $\wedge$  i = t then Some 1 else if t = 0  $\wedge$  i =*  
*e then Some 0 else None*

*in case po of* None  $\Rightarrow$  do {

*id*  $\leftarrow$  *destrci i s*;

*td*  $\leftarrow$  *destrci t s*;

*ed*  $\leftarrow$  *destrci e s*;

*let a = (case id of IFD v t e  $\Rightarrow$  v)*;

*let a = (case td of IFD v t e  $\Rightarrow$  min a v | -  $\Rightarrow$  a)*;

*let a = (case ed of IFD v t e  $\Rightarrow$  min a v | -  $\Rightarrow$  a)*;

*let ti = (case id of IFD v ti ei  $\Rightarrow$  if v = a then ti*

*else i | -  $\Rightarrow$  i)*;

*let tt = (case td of IFD v ti ei  $\Rightarrow$  if v = a then ti*

*else t | -  $\Rightarrow$  t)*;

*let te = (case ed of IFD v ti ei  $\Rightarrow$  if v = a then ti*

*else e | -  $\Rightarrow$  e)*;

*let fi = (case id of IFD v ti ei  $\Rightarrow$  if v = a then ei*

*else i | -  $\Rightarrow$  i)*;

*let ft = (case td of IFD v ti ei  $\Rightarrow$  if v = a then ei*

*else t | -  $\Rightarrow$  t)*;

*let fe = (case ed of IFD v ti ei  $\Rightarrow$  if v = a then ei*

*else e | -  $\Rightarrow$  e)*;

      (*tb*, *s*)  $\leftarrow$  *iteci-lu-code ti tt te s*;

      (*fb*, *s*)  $\leftarrow$  *iteci-lu-code fi ft fe s*;

      (*r*, *s*)  $\leftarrow$  *ifci a tb fb s*;

*cl*  $\leftarrow$  *hm-update* (*i*, *t*, *e*) *r* (*dcli s*);

*return* (*r*, *dcli-update* (*const cl*) *s*)

    }

  | *Some b*  $\Rightarrow$  *return* (*b*, *s*)

  | *Some b*  $\Rightarrow$  *return* (*b*, *s*)

}

**declare** *iteci-lu-code.simps*[code]

**lemma** *iteci-lu-code*[code-unfold]: *iteci-lu i t e s = iteci-lu-code i t e s*

**oops**

**lemma** *iteci-lu-rule*:

```

(mi.ite-impl-lu i t e bdd = Some (p,bdd'))  $\longrightarrow$ 
<is-bdd-impl bdd bddi>
  iteci-lu i t e bddi
< $\lambda(p_i, bdd_i'). is-bdd-impl bdd' bdd_i' * \uparrow(p_i=p) >_t$ 
apply (induction arbitrary: i t e bddi bdd p bdd' rule: mi.ite-impl-lu.fixp-induct)
  subgoal
    apply simp
    using option-admissible[where P=
       $\lambda((x1,x2),x3),x4) (r1,r2). \forall bddi.$ 
      <is-bdd-impl x4 bddi>
      iteci-lu x1 x2 x3 bddi
      < $\lambda r. case r of (p_i, bdd_i') \Rightarrow is-bdd-impl r2 bdd_i' * \uparrow(p_i = r1) >_t$ ]
    apply auto[1]
    apply (fo-rule subst[rotated])
    apply (assumption)
    by auto
  subgoal by simp
  subgoal
    apply clarify
    apply (clarsimp split: option.splits Option.bind-splits prod.splits)
  subgoal
    unfolding updS-def
    apply (subst iteci-lu.simps)
    apply (sep-auto)
    using bdd-hm-lookup-rule apply(blast)
    apply(sep-auto)
    apply(rule fi-rule)
    apply(rule param-optci-rule)
    apply(sep-auto)
    apply(sep-auto)
    apply(sep-auto)
    unfolding imp-to-meta
    apply(rule fi-rule)
    apply(rprems)
    apply(simp; fail)
    apply(sep-auto)
    apply(sep-auto)
    apply(rule fi-rule)
    apply(rprems)
    apply(simp; fail)
    apply(sep-auto)
    apply(sep-auto)
    unfolding updS-def by (sep-auto)
  subgoal
    apply(subst iteci-lu.simps)
    apply(sep-auto)
    using bdd-hm-lookup-rule apply(blast)
    apply(sep-auto)
    apply(rule fi-rule)

```

```

    apply(rule param-optci-rule)
    apply(sep-auto)
    apply(sep-auto)
    by (sep-auto)
  subgoal
    apply(subst iteci-lu.simps)
    apply(sep-auto)
    using bdd-hm-lookup-rule apply(blast)
    by(sep-auto)
  done
done

```

## 10.1 A standard library of functions

```
declare iteci-rule[THEN mp, sep-heap-rules]
```

```

definition notci e s  $\equiv$  do {
  (f,s)  $\leftarrow$  fci s;
  (t,s)  $\leftarrow$  tci s;
  iteci-lu e f t s
}

```

```

definition orci e1 e2 s  $\equiv$  do {
  (t,s)  $\leftarrow$  tci s;
  iteci-lu e1 t e2 s
}

```

```

definition andci e1 e2 s  $\equiv$  do {
  (f,s)  $\leftarrow$  fci s;
  iteci-lu e1 e2 f s
}

```

```

definition norci e1 e2 s  $\equiv$  do {
  (r,s)  $\leftarrow$  orci e1 e2 s;
  notci r s
}

```

```

definition nandci e1 e2 s  $\equiv$  do {
  (r,s)  $\leftarrow$  andci e1 e2 s;
  notci r s
}

```

```

definition biimpci a b s  $\equiv$  do {
  (nb,s)  $\leftarrow$  notci b s;
  iteci-lu a b nb s
}

```

```

definition xorci a b s  $\equiv$  do {
  (nb,s)  $\leftarrow$  notci b s;
  iteci-lu a nb b s
}

```

```

definition litci v bdd  $\equiv$  do {
  (t,bdd)  $\leftarrow$  tci bdd;
  (f,bdd)  $\leftarrow$  fci bdd;
}

```

```

    ifci v t f bdd
  }
definition tautci v bdd ≡ do {
  d ← destrci v bdd;
  return (d = TD)
}

```

## 10.2 Printing

The following functions are exported unverified. They are intended for BDD debugging purposes.

**partial-function**(heap) *serializeci* :: nat ⇒ bddi ⇒ ((nat × nat) × nat) list Heap **where**

```

serializeci p s = do {
  d ← destrci p s;
  (case d of
    IFD v t e ⇒ do {
      r ← serializeci t s;
      l ← serializeci e s;
      return (remdups (((p,t),1),((p,e),0)] @ r @ l))
    } |
    - ⇒ return []
  )
}

```

**declare** *serializeci.simps*[code]

**fun** *mapM* **where**

```

mapM f [] = return [] |
mapM f (a#as) = do {
  r ← f a;
  rs ← mapM f as;
  return (r#rs)
}

```

**definition** *liftM* f ma = do { a ← ma; return (f a) }

**definition** *sequence* = *mapM* id

**term** *liftM* (map f)

**lemma** *liftM* (map f) (*sequence* l) = *sequence* (map (*liftM* f) l)

**apply**(*induction* l)

**apply**(*simp* add: *sequence-def liftM-def*)

**apply**(*simp*)

**oops**

**fun** *string-of-nat* :: nat ⇒ string **where**

```

string-of-nat n = (if n < 10 then [char-of-nat (48 + n)]
  else string-of-nat (n div 10) @ [char-of-nat (48 + (n mod
10))])

```

**definition** *labelci* :: bddi ⇒ nat ⇒ (string × string × string) Heap **where**



```

labelci s n = do {
  d ← destrci n s;
  let son = string-of-nat n;
  let label = (case d of
    TD ⇒ "T" |
    FD ⇒ "F" |
    (IFD v -) ⇒ string-of-nat v);
  return (label, son, son @ "[label=" @ label @ "];
")
}

```

```

definition graphifyci1 bdd a ≡ do {
  let ((f,t),y) = a;
  let c = (string-of-nat f @ " -> " @ string-of-nat t);
  return (c @ (case y of 0 ⇒ "[style=dotted]" | Suc - ⇒ "")) @ " ";
")
}

```

**definition** trd = snd ∘ snd

**definition** fstp = apsnd fst

```

definition the-thing-By f l = (let
  nub = remdups (map fst l) in
  map (λe. (e, map snd (filter (λg. (f e (fst g))) l))) nub)

```

**definition** the-thing = the-thing-By (op =)

```

definition graphifyci :: string ⇒ nat ⇒ bddi ⇒ string Heap where
graphifyci name ep bdd ≡ do {
  s ← serializeci ep bdd;
  let e = map fst s;
  l ← mapM (labelci bdd) (rev (remdups (map fst e @ map snd e)));
  let grp = (map (λl. foldr (λa t. t @ a @ " ;") (snd l) "{rank=same;" @ "}")
") (the-thing (map fstp l)));
  e ← mapM (graphifyci1 bdd) s;
  let emptyhlp = (case ep of 0 ⇒ "F";
" | Suc 0 ⇒ "T";
" | - ⇒ "");
  return ("digraph " @ name @ " {
" @ concat (map trd l) @ concat grp @ concat e @ emptyhlp @ "}")
}

```

**end**

## 11 Collapsing the levels

```

theory Level-Collapse
imports Conc-Impl
begin

```

The theory up to this point is implemented in a way that separated the different aspects into different levels. This is highly beneficial for us, since it allows us to tackle the difficulties arising in small chunks. However, exporting this to the user would be highly impractical. Thus, this theory collapses all the different levels (i.e. refinement steps) and relates the computations in the heap monad to *boolfunc*.

**definition** *bddmi-rel*  $cs \equiv \{(a,c) \mid a \ b \ c. (a,b) \in \text{bf-ifex-rel} \wedge (c,b) \in \text{Rmi } cs\}$

**definition** *bdd-relator*  $:: (\text{nat } \text{boolfunc} \times \text{nat}) \text{ set} \Rightarrow \text{bddi} \Rightarrow \text{assn}$  **where**  
*bdd-relator*  $p \ s \equiv \exists_A cs. \text{is-bdd-impl } cs \ s \ * \ \uparrow(p \subseteq (\text{bddmi-rel } cs) \wedge \text{bdd-sane } cs) \ * \ \text{true}$

The *assn* predicate *bdd-relator* is the interface that is exposed to the user. (The contents of the definition are not exposed.)

**lemma** *bdd-relator-mono*[intro!]:  $q \subseteq p \Longrightarrow \text{bdd-relator } p \ s \Longrightarrow_A \text{bdd-relator } q \ s$   
**unfolding** *bdd-relator-def* **by** *sep-auto*

**lemma** *bdd-relator-absorb-true*[simp]:  $\text{bdd-relator } p \ s \ * \ \text{true} = \text{bdd-relator } p \ s$  **unfolding** *bdd-relator-def* **by** *simp*

**thm** *bdd-relator-def*[unfolded *bddmi-rel-def*, *simplified*]

**lemma** *join-hlp1*:  $\text{is-bdd-impl } a \ s \ * \ \text{is-bdd-impl } b \ s \Longrightarrow_A \text{is-bdd-impl } a \ s \ * \ \text{is-bdd-impl } b \ s \ * \ \uparrow(a = b)$

**apply** *clarsimp*

**apply**(*rule preciseD*[**where**  $p=s$  **and**  $R=\text{is-bdd-impl}$  **and**  $F=\text{is-bdd-impl } b \ s$  **and**  $F'=\text{is-bdd-impl } a \ s$ ])

**apply**(*rule is-bdd-impl-prec*)

**apply**(*unfold mod-and-dist*)

**apply**(*rule conjI*)

**apply** *assumption*

**apply**(*simp add: star-aci(2)*)

**done**

**lemma** *join-hlp*:  $\text{is-bdd-impl } a \ s \ * \ \text{is-bdd-impl } b \ s = \text{is-bdd-impl } b \ s \ * \ \text{is-bdd-impl } a \ s \ * \ \uparrow(a = b)$

**apply**(*rule ent-iffI*[rotated])

**apply**(*simp; fail*)

**apply**(*rule ent-trans*)

**apply**(*rule join-hlp1*)

**apply**(*simp; fail*)

**done**

**lemma** *add-true-asm*:

**assumes**  $\langle b \ * \ \text{true} \rangle \ p \ \langle a \rangle_t$

**shows**  $\langle b \rangle \ p \ \langle a \rangle_t$

**apply**(*rule cons-pre-rule*)

**prefer** 2

**apply**(*rule assms*)

**apply**(*simp add: ent-true-drop*)

**done**

```
lemma add-anything:  
  assumes  $\langle b \rangle p \langle a \rangle$   
  shows  $\langle b * x \rangle p \langle \lambda r. a r * x \rangle_t$   
proof –  
  note [sep-heap-rules] = assms  
  show ?thesis by sep-auto  
qed
```

```
lemma add-true:  
  assumes  $\langle b \rangle p \langle a \rangle_t$   
  shows  $\langle b * true \rangle p \langle a \rangle_t$   
  using assms add-anything[where  $x=true$ ] by force
```

**definition** *node-relator* **where** *node-relator*  $x y \longleftrightarrow x \in y$

*sep-auto* behaves sub-optimal when having  $(bf, bdd) \in \text{computed-pointer-relation}$  as assumption in our cases. Using *node-relator* instead fixes this behavior with a custom solver for *simp*.

**lemma** *node-relatorI*:  $x \in y \implies \text{node-relator } x y$  **unfolding** *node-relator-def* .

**lemma** *node-relatorD*:  $\text{node-relator } x y \implies x \in y$  **unfolding** *node-relator-def* .

**ML**(*fun TRY' tac = tac ORELSE' K all-tac*)

```
setup (map-theory-simpset (fn ctxt =>  
  ctxt addSolver (Simplifier.mk-solver node-relator  
    (fn ctxt => fn n =>  
      let  
        val tac =  
          resolve-tac ctxt @{thms node-relatorI} THEN'  
          REPEAT-ALL-NEW (resolve-tac ctxt @{thms Set.insertI1 Set.insertI2})  
        THEN'  
          TRY' (dresolve-tac ctxt @{thms node-relatorD} THEN' assume-tac ctxt)  
      in  
        SOLVED' tac n  
      end))  
    ))
```

This is the general form one wants to work with: if a function on the bdd is called with a set of already existing and valid pointers, the arguments to the function have to be in that set. The result is that one more pointer is the set of existing and valid pointers.

**thm** *iteci-rule*[*THEN mp*] *mi.ite-impl-R ifex-ite-rel-bf*

```
lemma iteci-rule[sep-heap-rules]:  
[[node-relator (ib, ic) rp; node-relator (tb, tc) rp; node-relator (eb, ec) rp]]  $\implies$ 
```

```

<bdd-relator rp s>
  iteci-lu ic tc ec s
< $\lambda(r,s'). \text{bdd-relator } (\text{insert } (\text{bf-ite } ib \text{ } tb \text{ } eb,r) \text{ } rp) \text{ } s'$ >
  apply(unfold bdd-relator-def node-relator-def)
  apply(intro norm-pre-ex-rule)
  apply(clarsimp)
  apply(unfold bddmi-rel-def)
  apply(drule (1) rev-subsetD)+
  apply(clarsimp)
  apply(drule (3) mi.ite-impl-lu-R[where ii=ic and ti=tc and ei=ec, unfolded
in-rel-def])
  apply(drule ospecD2)
  apply(clarsimp simp del: ifex-ite.simps)
  apply(rule cons-post-rule)
  apply(rule cons-pre-rule[rotated])
  apply(rule iteci-lu-rule[THEN mp, THEN add-true])
  apply(assumption)
  apply(sep-auto; fail)
  apply(clarsimp simp del: ifex-ite.simps)
  apply(rule ent-ex-postI)
  apply(subst ent-pure-post-iff)
  apply(rule conjI[rotated])
  apply(sep-auto; fail)
  apply(clarsimp simp del: ifex-ite.simps)
  apply(rule conjI[rotated])
  apply(force simp add: mi.les-def)
  apply(rule exI)
  apply(rule conjI)
  apply(erule (2) ifex-ite-opt-rel-bf[unfolded in-rel-def])
  apply assumption
done

```

**lemma** tci-rule[sep-heap-rules]:

```

<bdd-relator rp s>
  tci s
< $\lambda(r,s'). \text{bdd-relator } (\text{insert } (\text{bf-True},r) \text{ } rp) \text{ } s'$ >
  apply(unfold bdd-relator-def)
  apply(intro norm-pre-ex-rule)
  apply(clarsimp)
  apply(frul mi.Timpl-rule)
  apply(drule ospecD2)
  apply(clarify)
  apply(sep-auto)
  apply(unfold bddmi-rel-def)
  apply(clarsimp)
  apply(force simp add: mi.les-def)
done

```

**lemma** fci-rule[sep-heap-rules]:

```

<bdd-relator rp s>
  fci s
< $\lambda(r,s'). \text{ bdd-relator } (\text{insert } (\text{bf-False},r) \text{ rp}) s'$ >
  apply(unfold bdd-relator-def)
  apply(intro norm-pre-ex-rule)
  apply(clarsimp)
  apply(frule mi.Fimpl-rule)
  apply(drule ospecD2)
  apply(clarify)
  apply(sep-auto)
  apply(unfold bddmi-rel-def)
  apply(clarsimp)
  apply(force simp add: mi.les-def)
done

```

IFC/ifmi/ifci require that the variable order is ensured by the user. Instead of using ifci, a combination of litci and iteci has to be used.

```

lemma [sep-heap-rules]:
 $\llbracket (tb, tc) \in rp; (eb, ec) \in rp \rrbracket \implies$ 
<bdd-relator rp s>
  ifci v tc ec s
< $\lambda(r,s'). \text{ bdd-relator } (\text{insert } (\text{bf-if } v \text{ tb } eb,r) \text{ rp}) s'$ >

```

This probably doesn't hold.

**oops**

```

lemma notci-rule[sep-heap-rules]:
  assumes node-relator (tb, tc) rp
  shows <bdd-relator rp s> notci tc ec s < $\lambda(r,s'). \text{ bdd-relator } (\text{insert } (\text{bf-not } tb,r) \text{ rp}) s'$ >
  using assms
  by(sep-auto simp: notci-def)

```

```

lemma cirules1[sep-heap-rules]:
  assumes node-relator (tb, tc) rp node-relator (eb, ec) rp
  shows
    <bdd-relator rp s> andci tc ec s < $\lambda(r,s'). \text{ bdd-relator } (\text{insert } (\text{bf-and } tb \text{ eb},r) \text{ rp}) s'$ >
    <bdd-relator rp s> orci tc ec s < $\lambda(r,s'). \text{ bdd-relator } (\text{insert } (\text{bf-or } tb \text{ eb},r) \text{ rp}) s'$ >
    <bdd-relator rp s> biimpci tc ec s < $\lambda(r,s'). \text{ bdd-relator } (\text{insert } (\text{bf-biimp } tb \text{ eb},r) \text{ rp}) s'$ >
    <bdd-relator rp s> xorci tc ec s < $\lambda(r,s'). \text{ bdd-relator } (\text{insert } (\text{bf-xor } tb \text{ eb},r) \text{ rp}) s'$ >
  using assms
  by (sep-auto simp: andci-def orci-def biimpci-def xorci-def)+

```

```

lemma cirules2[sep-heap-rules]:

```

```

assumes node-relator (tb, tc) rp node-relator (eb, ec) rp
shows
  <bdd-relator rp s> nandci tc ec s < $\lambda(r,s')$ . bdd-relator (insert (bf-nand tb eb,r)
rp) s'>
  <bdd-relator rp s> norci tc ec s < $\lambda(r,s')$ . bdd-relator (insert (bf-nor tb eb,r)
rp) s'>
using assms
by(sep-auto simp: nandci-def norci-def)+

```

```

lemma litci-rule[sep-heap-rules]:
  <bdd-relator rp s> litci v s < $\lambda(r,s')$ . bdd-relator (insert (bf-lit v,r) rp) s'>
apply(unfold litci-def)
apply(subgoal-tac  $\wedge t$  ab bb. (* introducing some vars... *)
  <bdd-relator (insert (bf-False, ab) (insert (bf-True, t) rp)) bb * true>
  ifci v t ab bb
  < $\lambda r$ . case r of (r, x)  $\Rightarrow$  bdd-relator (insert (bf-lit v, r) rp) x>)
apply(sep-auto; fail)
apply(rename-tac tc fc sc)
apply(unfold bdd-relator-def[abs-def])
apply(clarsimp)
apply(intro norm-pre-ex-rule)
apply(clarsimp)
apply(unfold bddmi-rel-def)
apply(clarsimp simp only: bf-ifex-rel-consts-ensured)
apply(frule mi.IFimpl-rule)
  apply(rename-tac tc fc sc sm a aa b ba fm tm)
  apply(thin-tac (fm, Falseif)  $\in Rmi$  sm)
  apply(assumption)
  apply(assumption)
apply(clarsimp)
apply(drule ospecD2)
apply(clarify)
apply(sep-auto)
apply(force simp add: mi.les-def)
done

```

```

lemma tautci-rule[sep-heap-rules]:
shows node-relator (tb, tc) rp  $\Longrightarrow$  <bdd-relator rp s> tautci tc s < $\lambda r$ . bdd-relator
rp s *  $\uparrow(r \longleftrightarrow tb = \text{bf-True})$ >
apply(unfold node-relator-def)
apply(unfold tautci-def)
apply(unfold bdd-relator-def)
apply(intro norm-pre-ex-rule; clarsimp)
apply(unfold bddmi-rel-def)
apply(drule (1) rev-subsetD)
apply(clarsimp)
apply(rename-tac sm ti)
apply(frule (1) mi.DESTRimpl-rule; drule ospecD2; clarify)
apply(sep-auto split: ifex.splits)

```

**done**

**lemma** *emptyci-rule*[*sep-heap-rules*]:  
  **shows**  $\langle emp \rangle$  *emptyci*  $\langle \lambda r. bdd-relator \ \{ \} \ r \rangle$   
**by**(*sep-auto simp: bdd-relator-def*)

**lemmas** [*simp*] = *bf-ite-def*

Efficient comparison of two nodes.

**definition** *eqci*  $a \ b \equiv$  *return*  $(a = b)$

**lemma** *iteeq-rule*[*sep-heap-rules*]:  
   $\llbracket node-relator \ (xb, xc) \ rp; \ node-relator \ (yb, yc) \ rp \rrbracket \implies$   
   $\langle bdd-relator \ rp \ s \rangle$   
   $eqci \ xc \ yc$   
   $\langle \lambda r. \uparrow(r \longleftrightarrow xb = yb) \rangle_t$   
  **apply**(*unfold bdd-relator-def node-relator-def eqci-def*)  
  **apply**(*intro norm-pre-ex-rule*)  
  **apply**(*clarsimp*)  
  **apply**(*unfold bddmi-rel-def*)  
  **apply**(*drule* (1) *rev-subsetD*)  
  **apply**(*rule return-cons-rule*)  
  **apply**(*clarsimp*)  
  **apply**(*rule iffI*)  
  **using** *bf-ifex-eq mi.cmp-rule-eq* **apply**(*blast*)  
  **using** *bf-ifex-eq mi.cmp-rule-eq* **apply**(*blast*)  
**done**

**end**

## 12 Tests and examples

**theory** *BDD-Examples*  
**imports** *Level-Collapse*  
**begin**

Just two simple examples:

**lemma**  $\langle emp \rangle$  *do* {  
   $s \leftarrow emptyci$ ;  
   $(t,s) \leftarrow tci \ s$ ;  
   $tautci \ t \ s$   
}  $\langle \lambda r. \uparrow(r = True) \rangle_t$   
**by** *sep-auto*

**lemma**  $\langle emp \rangle$  *do* {  
   $s \leftarrow emptyci$ ;

```

(a,s) ← litci 0 s;
(b,s) ← litci 1 s;
(c,s) ← litci 2 s;
(t1i,s) ← orci a b s;
(t1,s) ← andci t1i c s;
(t2i1,s) ← andci a c s;
(t2i2,s) ← andci b c s;
(t2,s) ← orci t2i1 t2i2 s;
eqci t1 t2
} <↑>_t
by sep-auto

end

```

## References

- [1] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45. ACM, 1991.
- [2] M. Giorgino and M. Strecker. Correctness of pointer manipulating algorithms illustrated by a verified BDD construction. In *FM 2012: Formal Methods*, pages 202–216. Springer, 2012.
- [3] T. Nipkow. Boolean Expression Checkers. *Archive of Formal Proofs*, June 2014. [http://isa-afp.org/entries/Boolean\\_Expression\\_Checkers.shtml](http://isa-afp.org/entries/Boolean_Expression_Checkers.shtml), Formal proof development.
- [4] V. Ortner and N. Schirmer. BDD Normalisation. *Archive of Formal Proofs*, Feb. 2008. <http://isa-afp.org/entries/BDD.shtml>, Formal proof development.